

# Блокировки Блокировки в оперативной памяти



## Авторские права

© Postgres Professional, 2016–2025

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Игорь Гнатюк

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

## Использование материалов курса

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

## Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

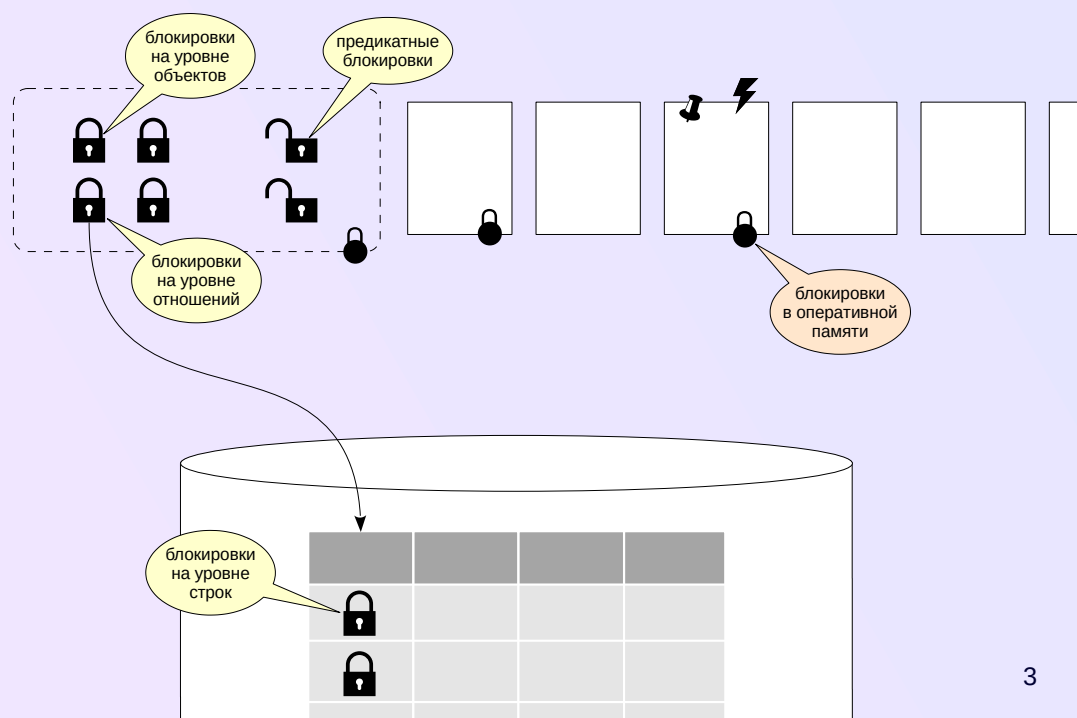
[edu@postgrespro.ru](mailto:edu@postgrespro.ru)

## Отказ от ответственности

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

Блокировки в памяти  
Мониторинг ожиданий

# Блокировки в памяти



Устанавливаются на очень короткое время

несколько инструкций процессора

Исключительный режим

Есть мониторинг

`pg_stat_activity`

Нет обнаружения взаимоблокировок

Цикл активного ожидания

используются атомарные инструкции процессора

Мы уже рассмотрели долговременные блокировки, действующие как правило до конца транзакции и поддерживающие множество режимов. Для защиты структур в оперативной памяти, разделяемой несколькими процессами, используются более простые (и дешевые в смысле накладных расходов) блокировки.

Самые простые из них — спин-блокировки или спинлоки (spinlock). Они предназначены для захвата на очень короткое время (несколько инструкций процессора) и защищают отдельные поля структур данных от одновременного изменения.

Спин-блокировки реализуются на основе атомарных инструкций процессора, например, `compare-and-swap` ([https://ru.wikipedia.org/wiki/Сравнение\\_с\\_обменом](https://ru.wikipedia.org/wiki/Сравнение_с_обменом)).

Они захватываются только в исключительном режиме. Если блокировка занята, выполняется цикл активного ожидания — команда повторяется до тех пор, пока не выполнится успешно. Это имеет смысл, поскольку спин-блокировки применяются только в тех случаях, когда вероятность конфликта очень мала. Существует и мониторинг отслеживания ожиданий, связанных со спин-блокировками.

Однако спин-блокировки не обеспечивают обнаружения взаимоблокировок — за этим следят разработчики кода PostgreSQL.

Устанавливаются на короткое время

обычно доли секунды

Исключительный и разделяемый режимы

Есть мониторинг

`pg_stat_activity`

Нет обнаружения взаимоблокировок

«Нечестная» очередь

разделяемая блокировка позволяет читающим процессам проходить без очереди

Следом идут так называемые легкие блокировки (lightweight locks, lwlocks).

Их захватывают на короткое время, которое требуется для работы со структурой данных (например, с хеш-таблицей или списком указателей). Как правило, легкая блокировка удерживается недолго, но в процессе ее удержания могут выполняться операции ввода-вывода, так что иногда время может оказаться и значительным.

Используются два режима блокирования: исключительный (для записи) и разделяемый (для чтения).

Поддерживается очередь ожидания. Однако пока блокировка удерживается в разделяемом режиме, другие читающие процессы проходят вне очереди. В системах с высокой степенью параллельности и большой нагрузкой это может приводить к непредсказуемо долгим ожиданиям процессов, которым требуется изменять данные (<https://postgrespro.ru/list/thread-id/2400193>).

Проверка взаимоблокировок не выполняется, как и для спин-блокировок. Легкие блокировки также имеют средства для мониторинга.

## Закрепление буфера

Устанавливается на время работы с буфером

возможно, длительное

Разделяемый режим

Есть мониторинг

`pg_stat_activity`

`pg_buffercache`

Нет обнаружения взаимоблокировок

Пассивное ожидание

но обычно закрепленный буфер пропускается

Еще один вид блокировки, который мы уже рассматривали в теме «Буферный кеш» модуля «Журналирование» — закрепление буфера (buffer pin).

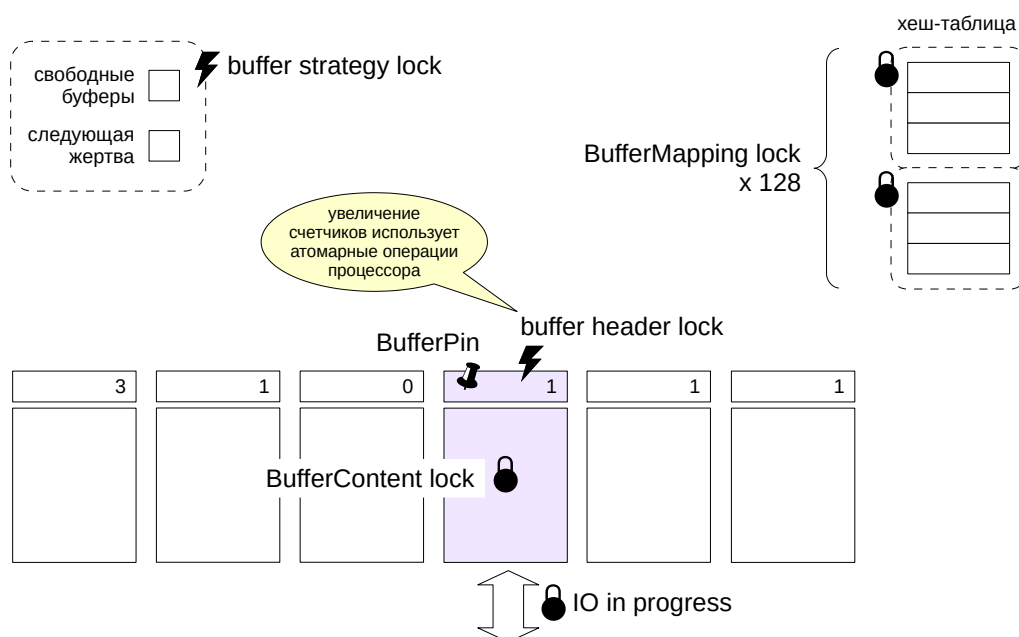
Для каждого буфера ведется список работающих с ним процессов. Если буфер закреплен, в нем запрещены некоторые действия. Например, в страницу можно вставить новую строку (которую остальные процессы не увидят благодаря многоверсионности), но нельзя заменить одну страницу на другую.

Как правило, процессы пропускают закрепленный буфер и выбирают другой, чтобы не ждать снятия закрепления. Но в некоторых случаях, когда требуется именно этот буфер, запрашивается легкая блокировка в исключительном режиме. При необходимости процесс ждет, пока все остальные процессы закончат работу с буфером. Ожидание при этом пассивное: система «будит» ожидающий процесс, когда закрепление снимается.

Взаимоблокировки невозможны, за этим следят разработчики PostgreSQL.

Ожидания, связанные с закреплением, доступны для мониторинга через представление `pg_stat_activity`. Текущее количество закреплений буфера показывает расширение `pg_buffercache`.

# Пример: буферный кеш



7

Чтобы получить некоторое (неполное) представление о том, как и где используются блокировки, рассмотрим пример буферного кеша.

Чтобы обратиться к хеш-таблице, содержащей ссылки на буферы, процесс должен захватить легкую блокировку `BufferMapping lock` в разделяемом режиме, а если таблицу требуется изменять — то в исключительном режиме. Чтобы увеличить гранулярность, эта блокировка устроена как *транш*, состоящий из 128 отдельных блокировок, каждая из которых защищает свою часть хеш-таблицы.

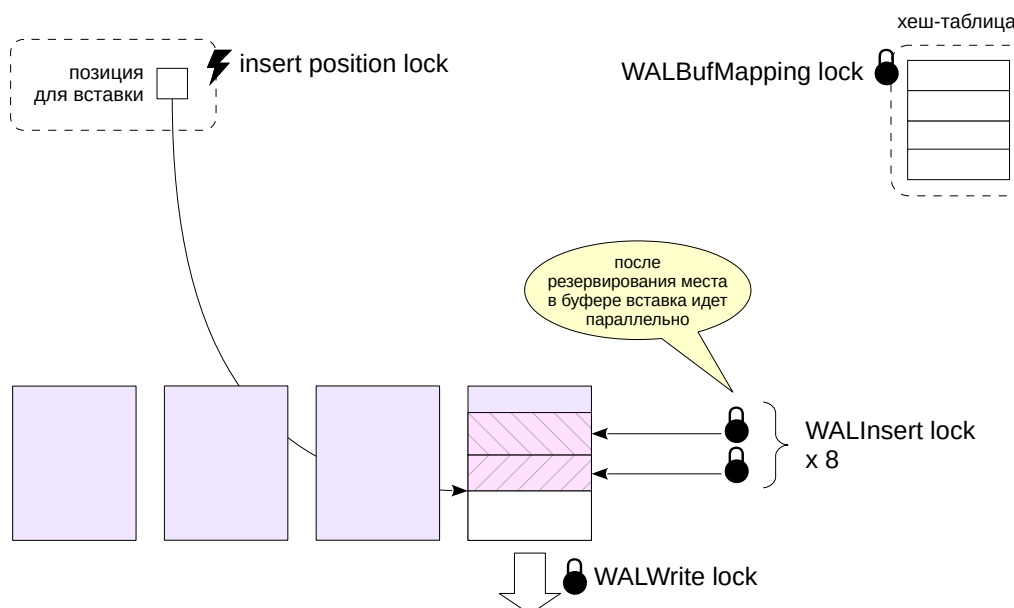
Доступ к заголовку буфера процесс получает с помощью спин-блокировки. Отдельные операции (такие, как увеличение счетчика) могут выполняться и без явных блокировок, с помощью атомарных инструкций процессора.

Чтобы прочитать содержимое буфера, требуется блокировка `BufferContent lock`. Обычно она захватывается только для чтения указателей на версии строк, а дальше достаточно защиты, предоставляемой закреплением буфера. Для изменения содержимого буфера эта блокировка захватывается в исключительном режиме.

При чтении буфера с диска (или записи на диск) в его заголовке устанавливается признак `IO in progress`, который сигнализирует другим процессам, что страница читается — они могут встать в очередь, если им тоже нужна та же самая страница.

Указатели на свободные буферы и на следующую жертву защищены одной спин-блокировкой `buffer strategy lock`.

## Пример: буферы журнала



8

Еще один пример: буферы журнала.

Для журнального кеша тоже используется хеш-таблица, содержащая отображение страниц в буферы. В отличие от хеш-таблицы буферного кеша, эта хеш-таблица защищена единственной легкой блокировкой WALBufMapping lock, поскольку размер журнального кеша меньше (обычно 1/32 от буферного кеша) и обращение к его буферам более упорядочено.

Запись страниц на диск защищена легкой блокировкой WALWrite lock, чтобы только один процесс одновременно мог выполнять эту операцию.

Чтобы создать журнальную запись, процесс должен сначала зарезервировать место в странице. Для этого он захватывает спин-блокировку insert position lock. После того, как место зарезервировано, процесс копирует содержимое своей записи в отведенное место. Эта операция может выполняться несколькими процессами одновременно, для чего запись защищена *траншем* из восьми легких блокировок WALInsert lock.

Здесь представлены не все блокировки, имеющие отношение к журналу предзаписи, но эта и предыдущая иллюстрации должны дать некоторое представление об использовании блокировок в оперативной памяти.



Ожидания в `pg_stat_activity` и семплирование

Типы ожиданий

Когда процесс ожидает чего-либо,  
этот факт отражается в представлении `pg_stat_activity`

`wait_event_type` — тип ожидания

`wait_event` — имя конкретного ожидания

Информация может быть неполной

охвачены не все места в коде, в которых могут быть ожидания

Информация только на текущий момент

единственный способ получить картину во времени — семплирование  
достоверная картина только при большом числе измерений

Для мониторинга ожиданий используется представление `pg_stat_activity`. Когда процесс (системный или обслуживающий) не может выполнять свою работу и ждет чего-либо, это ожидание можно увидеть в представлении. Столбец `wait_event_type` показывает тип ожидания, а столбец `wait_event` — имя конкретного ожидания.

Следует учитывать, что представление показывает только те ожидания, которые соответствующим образом обрабатываются в исходном коде. Если представление не показывает ожидание, это вообще говоря не означает со 100-процентной вероятностью, что процесс действительно ничего не ждет.

К сожалению, единственная доступная информация об ожиданиях — информация на текущий момент. Никакой накопленной статистики не ведется. Единственный способ получить картину ожиданий во времени — семплирование состояния представления с определенным интервалом. Встроенных средств для этого не предусмотрено, но можно использовать расширения, например, `pg_wait_sampling`.

[https://github.com/postgrespro/pg\\_wait\\_sampling](https://github.com/postgrespro/pg_wait_sampling)

При семплировании надо учитывать его вероятностный характер. Чтобы получить более или менее достоверную картину, число измерений должно быть достаточно высоко. Поэтому семплирование с низкой частотой не даст достоверной картины, а повышение частоты приводит к увеличению накладных расходов. По той же причине семплирование бесполезно для анализа короткоживущих сеансов.

## Блокировки

блокировки объектов  
легкие блокировки  
закрепление буфера

pg\_stat\_activity.wait\_event\_type

Lock  
LWLock  
BufferPin

## Другие ожидания

ввод-вывод  
получение данных от другого процесса  
получение данных от клиента  
ожидание в модуле расширения  
«безделье»

IO  
IPC  
Client  
Extension  
Activity, Timeout

Все остальное — неучтенное время

11

Все ожидания можно разделить на несколько типов. Ожидания рассмотренных блокировок составляют большую категорию: ожидание блокировок объектов (значение Lock в столбце wait\_event\_type), ожидание легких блокировок (LWLock) и ожидание закрепленного буфера (BufferPin).

Но процессы могут ожидать и другие события. Ожидания ввода-вывода (IO) возникают, когда процессу требуется записать или прочитать данные. Процесс может ждать данные, необходимые для работы, от клиента (Client) или от другого процесса (IPC).

Расширения могут регистрировать свои специфические ожидания (Extension).

Бывают ситуации, когда процесс просто не выполняет полезной работы. К этой категории относится ожидание фоновых процессов в своем основном цикле (Activity), ожидание таймера (Timeout). Как правило, такие ожидания «нормальны» и не говорят о каких-либо проблемах.

Тип ожидания сопровождается именем конкретного ожидания:

<https://postgrespro.ru/docs/postgresql/16/monitoring-stats#WAIT-EVENT-TABLE>

Если имя ожидания не определено, процесс не находится в состоянии ожидания. Такое время следует считать неучтенным, так как на самом деле неизвестно, что именно происходит в этот момент.

В следующей демонстрации для замедления ввода-вывода мы используем файловую систему FUSE (<https://github.com/libfuse/libfuse>) и проект slowfs, построенный с ее помощью (<https://github.com/nirs/slowfs>).

## Мониторинг ожиданий

Текущие ожидания можно посмотреть в представлении `pg_stat_activity`, которое показывает информацию о работающих процессах. Выберем только часть полей:

```
=> SELECT pid, backend_type, wait_event_type, wait_event
FROM pg_stat_activity;
```

pid	backend_type	wait_event_type	wait_event
28375	autovacuum launcher	Activity	AutoVacuumMain
28376	logical replication launcher	Activity	LogicalLauncherMain
28420	client backend		
28372	background writer	Activity	BgWriterMain
28371	checkpointer	Activity	CheckpointerMain
28374	walwriter	Activity	WalWriterMain

(6 rows)

Пустые значения говорят о том, что процесс ничего не ждет и выполняет полезную работу.

Чтобы получить более или менее полную картину ожиданий процесса, требуется выполнять семплирование с некоторой частотой. Воспользуемся расширением `pg_wait_sampling`.

Расширение уже установлено из пакета в ОС виртуальной машины курса, но необходимо внести в конфигурационный параметр `shared_preload_libraries` название загружаемой библиотеки расширения. Применение этого параметра требует перезагрузки сервера.

```
=> ALTER SYSTEM SET shared_preload_libraries = 'pg_wait_sampling';
```

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 main restart
```

```
student$ psql
```

Теперь создадим расширение в базе данных.

```
=> CREATE DATABASE locks_memory;
```

```
CREATE DATABASE
```

```
=> \c locks_memory
```

You are now connected to database "locks\_memory" as user "student".

```
=> CREATE EXTENSION pg_wait_sampling;
```

```
CREATE EXTENSION
```

Расширение позволяет просмотреть некоторую историю ожиданий, которая хранится в кольцевом буфере. Но интереснее увидеть профиль ожиданий — накопленную статистику за все время работы.

Подождем несколько секунд и заглянем в профиль...

```
=> SELECT * FROM pg_wait_sampling_profile ORDER BY 1;
```

pid	event_type	event	queryid	count
28577	Activity	CheckpointerMain	0	926
28578	Activity	BgWriterMain	0	926
28580	IO	WALSync	0	28
28580	Activity	WalWriterMain	0	898
28581	Activity	AutoVacuumMain	0	926
28583	Activity	LogicalLauncherMain	0	926
28628	IO	WALSync	0	1
28628	IO	VersionFileSync	0	1
28628	IO	RelationMapReplace	0	3
28628	Client	ClientRead	0	8
28628	IO	DataFileRead	0	1
28678	Client	ClientRead	0	301

(12 rows)

Поскольку за прошедшее после запуска сервера время ничего не происходило, основные ожидания относятся к типу `Activity` (служебные процессы ждут, пока появится работа) и `Client` (`psql` ждет, пока пользователь пришлет запрос).

Строки с пустыми значениями `event_type` и `event` фиксируют ситуации, когда процесс ничего не ожидает (но работает и занимает процессорное время). За отображение таких строк отвечает параметр `pg_wait_sampling.sample_cpu`:

```
=> SHOW pg_wait_sampling.sample_cpu;
```

```
ERROR: unrecognized configuration parameter "pg_wait_sampling.sample_cpu"
```

С установками по умолчанию частота семплирования — 100 раз в секунду. Поэтому, чтобы оценить длительность ожиданий в секундах, значение count надо делить на 100.

Чтобы понять, к какому процессу относятся ожидания, добавим к запросу представление pg\_stat\_activity:

```
=> SELECT p.pid, a.backend_type, a.application_name AS app, p.event_type, p.event, p.count
FROM pg_wait_sampling_profile p
LEFT JOIN pg_stat_activity a ON p.pid = a.pid
ORDER BY p.pid, p.count DESC;
```

pid	backend_type	app	event_type	event	count
28577	checkpointer		Activity	CheckpointerMain	940
28578	background writer		Activity	BgWriterMain	940
28580	walwriter		Activity	WalWriterMain	912
28580	walwriter		IO	WALSync	28
28581	autovacuum launcher		Activity	AutoVacuumMain	940
28583	logical replication launcher		Activity	LogicalLauncherMain	940
28628			Client	ClientRead	8
28628			IO	RelationMapReplace	3
28628			IO	WALSync	1
28628			IO	VersionFileSync	1
28628			IO	DataFileRead	1
28678	client backend	psql	Client	ClientRead	315

(12 rows)

Готовимся дать нагрузку с помощью pgbench и наблюдать, как изменится картина.

```
student$ pgbench -i locks_memory
```

```
dropping old tables...
NOTICE: table "pgbench_accounts" does not exist, skipping
NOTICE: table "pgbench_branches" does not exist, skipping
NOTICE: table "pgbench_history" does not exist, skipping
NOTICE: table "pgbench_tellers" does not exist, skipping
creating tables...
generating data (client-side)...
100000 of 100000 tuples (100%) done (elapsed 0.33 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 0.63 s (drop tables 0.00 s, create tables 0.02 s, client-side generate 0.40 s,
vacuum 0.10 s, primary keys 0.11 s).
```

Сбрасываем собранный профиль в ноль и запускаем тест на 30 секунд в отдельном процессе. Одновременно будем смотреть, как изменяется профиль.

```
=> SELECT pg_wait_sampling.reset_profile();
```

```
pg_wait_sampling_reset_profile
-----
```

(1 row)

```
student$ pgbench -T 30 locks_memory
```

```
=> SELECT p.pid, a.backend_type, a.application_name AS app, p.event_type, p.event, p.count
FROM pg_wait_sampling_profile p
LEFT JOIN pg_stat_activity a ON p.pid = a.pid
WHERE a.application_name = 'pgbench'
ORDER BY p.pid, p.count DESC;
```

pid	backend_type	app	event_type	event	count
29064	client backend	pgbench	IO	WALSync	151
29064	client backend	pgbench	Client	ClientRead	23

(2 rows)

```
=> \g
```

pid	backend_type	app	event_type	event	count
29064	client backend	pgbench	I0	WALSync	1060
29064	client backend	pgbench	Client	ClientRead	153
29064	client backend	pgbench	I0	WALWrite	3
29064	client backend	pgbench	LWLock	WALWrite	2

(4 rows)

=> \g

pid	backend_type	app	event_type	event	count
29064	client backend	pgbench	I0	WALSync	1960
29064	client backend	pgbench	Client	ClientRead	283
29064	client backend	pgbench	I0	WALWrite	5
29064	client backend	pgbench	LWLock	WALWrite	3

(4 rows)

Ожидания процесса pgbench будут получаться разными в зависимости от конкретной системы. В нашем случае с большой вероятностью будет представлено ожидание записи и синхронизации журнала (IO/WALSync, IO/WALWrite).

```
pgbench (16.2 (Ubuntu 16.2-1ubuntu4))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 3870
number of failed transactions: 0 (0.000%)
latency average = 7.752 ms
initial connection time = 3.439 ms
tps = 128.997893 (without initial connection time)
```

## Легкие блокировки

Всегда нужно помнить, что отсутствие какого-либо ожидания при семплировании не говорит о том, что ожидания не было. Если оно было короче, чем период семплирования (сотая часть секунды в нашем примере), то могло просто не попасть в выборку.

Поэтому легкие блокировки скорее всего не появились в профиле — но появятся, если собирать данные в течении длительного времени.

Чтобы гарантированно увидеть их, подключимся к кластеру slow с замедленной файловой системой: в ней любая операция ввода-вывода будет занимать 1/10 секунды.

```
student$ sudo pg_ctlcluster 16 main stop
```

```
student$ sudo pg_ctlcluster 16 slow start
```

Еще раз сбросим профиль и дадим нагрузку.

=> \c

You are now connected to database "locks\_memory" as user "student".

```
=> SELECT pg_wait_sampling_reset_profile();
```

```
pg_wait_sampling_reset_profile
```

(1 row)

```
student$ pgbench -T 30 locks_memory
```

```
=> SELECT p.pid, a.backend_type, a.application_name AS app, p.event_type, p.event, p.count
FROM pg_wait_sampling_profile p
LEFT JOIN pg_stat_activity a ON p.pid = a.pid
WHERE a.application_name = 'pgbench'
ORDER BY p.pid, p.count DESC;
```

pid	backend_type	app	event_type	event	count
29452	client backend	pgbench	LWLock	WALWrite	180

(1 row)

=> \g

pid	backend_type	app	event_type	event	count
29491	client backend	pgbench	I0	WALWrite	819
29491	client backend	pgbench	LWLock	WALWrite	141
29491	client backend	pgbench	I0	WALSync	57
29491	client backend	pgbench	I0	DataFileExtend	20
29491	client backend	pgbench	Client	ClientRead	4
29491	client backend	pgbench	I0	DataFileRead	3

(6 rows)

=> \g

pid	backend_type	app	event_type	event	count
29491	client backend	pgbench	I0	WALWrite	1908
29491	client backend	pgbench	LWLock	WALWrite	162
29491	client backend	pgbench	I0	WALSync	122
29491	client backend	pgbench	I0	DataFileExtend	20
29491	client backend	pgbench	Client	ClientRead	7
29491	client backend	pgbench	I0	DataFileRead	5

(6 rows)

Теперь основное ожидание процесса pgbench связано с вводом-выводом, точнее с записью журнала, которая выполняется в синхронном режиме при каждой фиксации. Поскольку (вспомним слайд презентации) запись журнала на диск защищена легкой блокировкой WALWriteLock, она также присутствует в профиле.

```
pgbench (16.2 (Ubuntu 16.2-1ubuntu4))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 60
number of failed transactions: 0 (0.000%)
latency average = 504.396 ms
initial connection time = 43.501 ms
tps = 1.982567 (without initial connection time)
```

Блокировки в оперативной памяти реализуются по-разному

спин-блокировки, легкие блокировки, закрепление буфера  
относительно короткое время и облегченная инфраструктура

Мониторинг текущих ожиданий с помощью  
представления `pg_stat_activity`

семплирование для получения картины во времени



1. Открытый курсор удерживает закрепление буфера, чтобы чтение следующей строки выполнялось быстрее. Убедитесь в этом с помощью расширения `pg_buffercache`.
2. Откройте курсор по таблице и, не закрывая его, выполните очистку таблицы (`VACUUM`). Будет ли очистка ждать освобождения закрепления буфера?
3. Повторите эксперимент, выполнив очистку с заморозкой (`VACUUM FREEZE`). Убедитесь, что в профиль ожиданий обслуживающего процесса попало ожидание закрепления буфера.

1. Расширение `pg_buffercache` было рассмотрено в модуле «Журналирование», тема «Буферный кеш».

<https://postgrespro.ru/docs/postgresql/16/pgbuffercache.html>

## 1. Закрепление буфера при открытом курсоре

Сначала выполняем подготовительные действия.

Устанавливаем необходимые расширения:

```
=> ALTER SYSTEM SET shared_preload_libraries = 'pg_wait_sampling';
```

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 main restart
```

```
student$ psql
```

```
=> CREATE DATABASE locks_memory;
```

```
CREATE DATABASE
```

```
=> \c locks_memory
```

You are now connected to database "locks\_memory" as user "student".

```
=> CREATE EXTENSION pg_wait_sampling;
```

```
CREATE EXTENSION
```

```
=> CREATE EXTENSION pg_buffercache;
```

```
CREATE EXTENSION
```

Таблица, как в предыдущих практиках:

```
=> CREATE TABLE accounts(acc_no integer, amount numeric);
```

```
CREATE TABLE
```

```
=> INSERT INTO accounts VALUES (1,1000.00),(2,2000.00),(3,3000.00);
```

```
INSERT 0 3
```

Изменим остаток по счету 2 — в строке появится мертвая версия строки:

```
=> UPDATE accounts SET amount = 2500 WHERE acc_no = 2;
```

```
UPDATE 1
```

Начинаем транзакцию, открываем курсор и выбираем одну строку.

```
=> BEGIN;
```

```
BEGIN
```

```
=> DECLARE c CURSOR FOR SELECT * FROM accounts;
```

```
DECLARE CURSOR
```

```
=> FETCH c;
```

```
acc_no | amount  
-----+-----  
      1 | 1000.00  
(1 row)
```

Проверим, закреплен ли буфер:

```
=> SELECT * FROM pg_buffercache  
WHERE relfilenode = pg_relation_filenode('accounts') AND relforknumber = 0 \gx
```

```
-[ RECORD 1 ]-----  
bufferid      | 1854  
relfilenode   | 16416  
reltablespace | 1663  
reldatabase   | 16390  
relforknumber | 0  
relblocknumber | 0  
isdirty       | t  
usagecount    | 5  
pinning_backends | 1
```

Да, pinning\_backends = 1.

## 2. Очистка закрепленного буфера

Выполним очистку:

```
student$ psql locks_memory
```

```
=> VACUUM VERBOSE accounts;
```

```
INFO: vacuuming "locks_memory.public.accounts"
INFO: finished vacuuming "locks_memory.public.accounts": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total)
tuples: 0 removed, 4 remain, 0 are dead but not yet removable
tuples missed: 1 dead from 1 pages not removed due to cleanup lock contention
removable cutoff: 749, which was 0 XIDs old when operation ended
new relfrozenxid: 747, which is 1 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers
removed
avg read rate: 0.000 MB/s, avg write rate: 64.744 MB/s
buffer usage: 22 hits, 0 misses, 3 dirtied
WAL usage: 4 records, 3 full page images, 24911 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO: vacuuming "locks_memory.pg_toast.pg_toast_16416"
INFO: finished vacuuming "locks_memory.pg_toast.pg_toast_16416": index scans: 0
pages: 0 removed, 0 remain, 0 scanned (100.00% of total)
tuples: 0 removed, 0 remain, 0 are dead but not yet removable
removable cutoff: 749, which was 0 XIDs old when operation ended
new relfrozenxid: 749, which is 3 XIDs ahead of previous value
frozen: 0 pages from table (100.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (100.00% of total) had 0 dead item identifiers
removed
avg read rate: 84.005 MB/s, avg write rate: 0.000 MB/s
buffer usage: 24 hits, 1 misses, 0 dirtied
WAL usage: 1 records, 0 full page images, 188 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

Как мы видим, мертвая строка не была очищена: ...tuples missed: 1 dead from 1 pages not removed due to cleanup lock contention.

Если буфер закреплен, из страницы запрещено удалять версии строк. Но очистка не ждет, пока буфер освободится — строка будет очищена при следующем сеансе очистки.

## 3. Заморозка закрепленного буфера

Выполняем очистку с заморозкой:

```
=> VACUUM FREEZE VERBOSE accounts;
```

Очистка зависит до закрытия курсора. При явно запрошенной заморозке нельзя оставить необработанной ни одну страницу, не отмеченную в карте заморозки — иначе невозможно уменьшить максимальный возраст незамороженных транзакций в pg\_class.relfrozenxid.

```
=> SELECT age(relfrozenxid) FROM pg_class WHERE oid = 'accounts'::regclass;
```

```
age
-----
2
(1 row)
```

```
=> COMMIT;
```

```
COMMIT
```

```

INFO: aggressively vacuuming "locks_memory.public.accounts"
INFO: finished vacuuming "locks_memory.public.accounts": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total)
tuples: 1 removed, 3 remain, 0 are dead but not yet removable
removable cutoff: 749, which was 0 XIDs old when operation ended
new relfrozenxid: 749, which is 2 XIDs ahead of previous value
frozen: 1 pages from table (100.00% of total) had 3 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers
removed
avg read rate: 0.000 MB/s, avg write rate: 0.002 MB/s
buffer usage: 9 hits, 0 misses, 1 dirtied
WAL usage: 4 records, 1 full page images, 8586 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 5.09 s
INFO: aggressively vacuuming "locks_memory.pg_toast.pg_toast_16416"
INFO: finished vacuuming "locks_memory.pg_toast.pg_toast_16416": index scans: 0
pages: 0 removed, 0 remain, 0 scanned (100.00% of total)
tuples: 0 removed, 0 remain, 0 are dead but not yet removable
removable cutoff: 749, which was 0 XIDs old when operation ended
frozen: 0 pages from table (100.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (100.00% of total) had 0 dead item identifiers
removed
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 3 hits, 0 misses, 0 dirtied
WAL usage: 0 records, 0 full page images, 0 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM

```

```
=> SELECT age(relfrozenxid) FROM pg_class WHERE oid = 'accounts'::regclass;
```

```

age
-----
  0
(1 row)

```

Профиль ожиданий:

```

=> SELECT p.pid, a.backend_type, a.application_name AS app, p.event_type, p.event, p.count
FROM pg_wait_sampling_profile p
LEFT JOIN pg_stat_activity a ON p.pid = a.pid
WHERE event_type = 'BufferPin'
ORDER BY p.pid, p.count DESC;

```

pid	backend_type	app	event_type	event	count
59711	client backend	psql	BufferPin	BufferPin	499

(1 row)

Тип ожидания BufferPin говорит о том, что очистка ждала освобождения буфера.