

# データベースの古くて新しいボトルネック Lock & WAL

2011年10月29日

**NEC** サービスプラットフォーム研究所

堀川 隆

人と地球にやさしい情報社会を  
イノベーションで実現する  
グローバルリーディングカンパニー

NECグループビジョン2017

# Q. Lock ?

---

## “Lock” で真っ先に思い浮かべるもの

- SQLレベルのlock  
select \* from foo where bar=baz for update;
- Postgres内部で使われているlock  
LWLockAcquire( foo, LW\_EXCLUSIVE); LWLockRelease(foo);
- System call (OS管理オブジェクトのlock)  
flock(foo, barMODE );
- System call (排他制御)  
sem\_wait(&foo); sem\_post(&foo);  
pthread\_mutex\_lock(&bar);
- 機械語命令レベルのlock  
lock; cmpxchgl foo,bar
- その他  
What ?
- (+ Lock は知らない)

# WAL

---

## 種々の側面

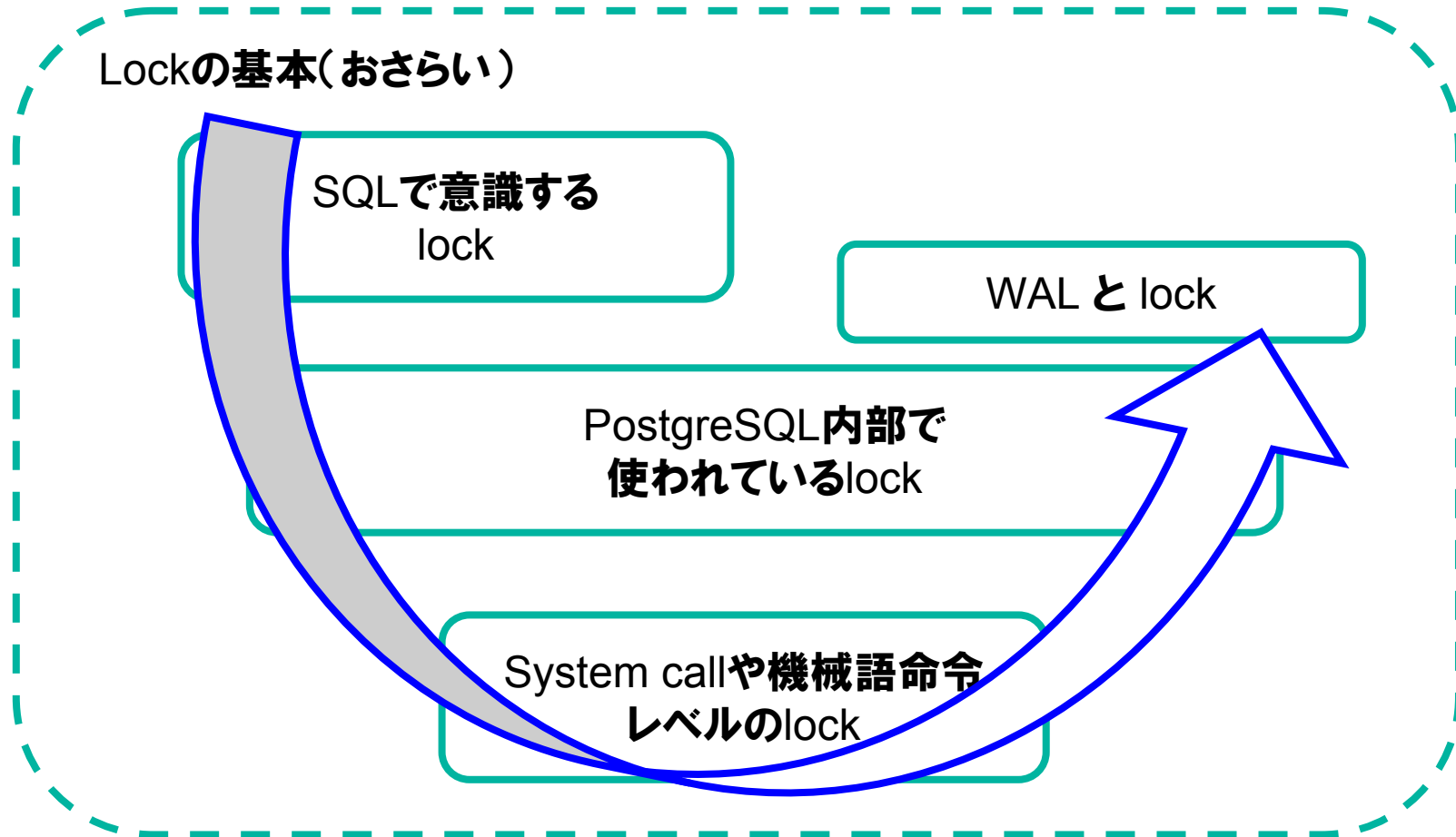
- データ・フォーマット(何を書き込んでいるか)
- 通常動作における振舞い
- 性能への影響(オーバーヘッド)
- リカバリ時の動作
- レプリケーション関連

## 既存資料

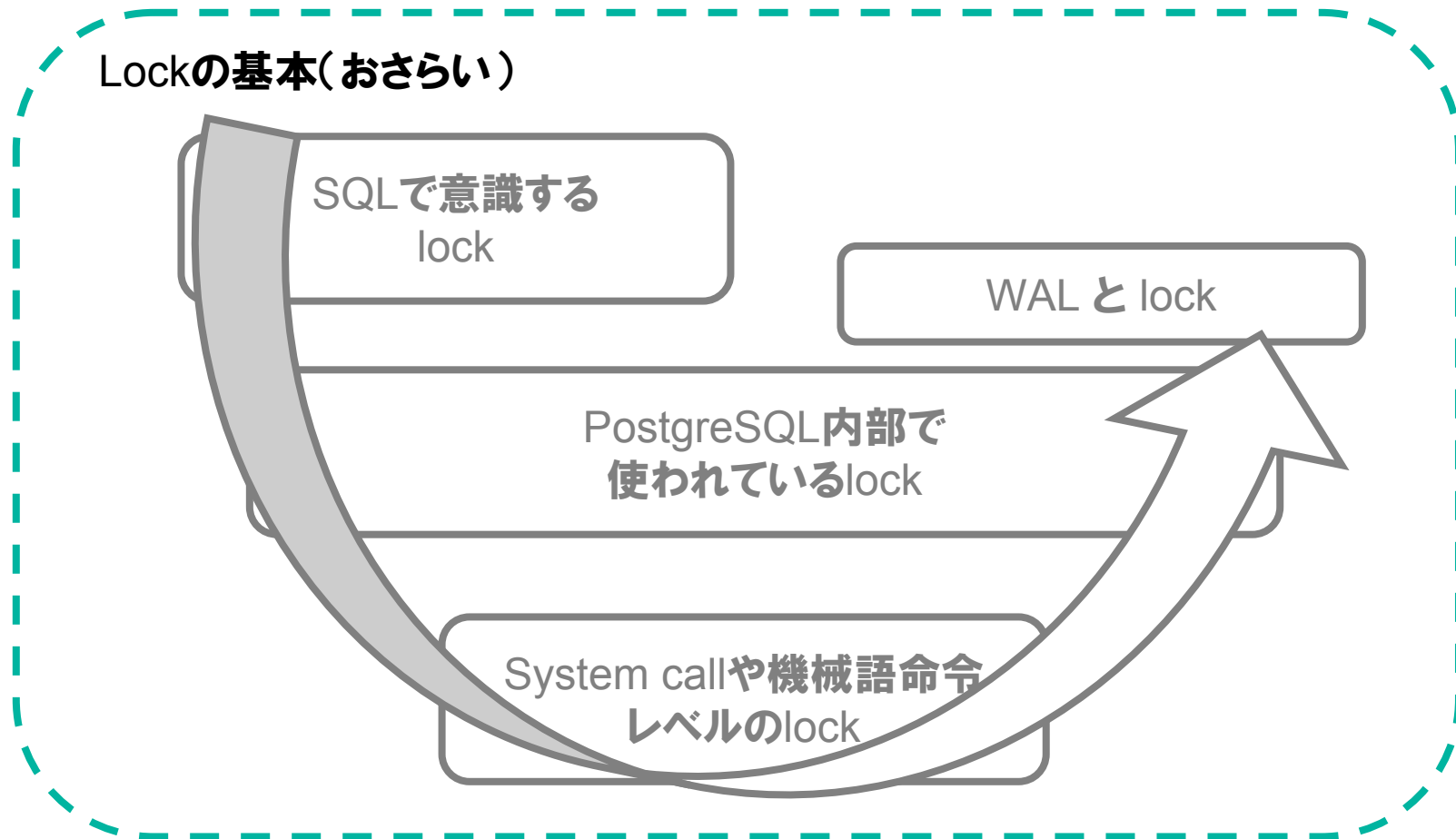
- Suzuki Hironobu @ InterDB,  
2-09 WALとは(Write Ahead Logging), 2-10 WALの詳細  
<http://www.interdb.jp/techinfo/postgresql/p-2-09.html>  
<http://www.interdb.jp/techinfo/postgresql/p-2-10.html>
- 坂田 哲夫氏, Logのしくみ, PostgreSQLのLogのしくみ(三訂版)  
[http://www.postgresql.jp/wg/shikumi/shikumi\\_archive/shikumi\\_archive\\_files/20040721105406-shikumi\\_040726\\_logging1.pdf](http://www.postgresql.jp/wg/shikumi/shikumi_archive/shikumi_archive_files/20040721105406-shikumi_040726_logging1.pdf)  
[http://www.postgresql.jp/wg/shikumi/shikumi\\_archive/postgresql13/logging0902.pdf](http://www.postgresql.jp/wg/shikumi/shikumi_archive/postgresql13/logging0902.pdf)

→ これまでとは**違う視点**からWALを眺めてみる

# Agenda



# Agenda



# Terminology (Lock, mutex, and latch)

## Lock

1. データやデバイスなどのリソースへのアクセス制限を課す同期機構
2. SQLレベルで意識できる排他制御

リソース利用を排他制御する点を意識した表現

## Mutex (MUTual EXclusionの略)

- クリティカルセクションでアトミック性を確保するための同期機構の一種  
Mutex獲得に失敗したスレッドはblockされる(sleepする)

排他制御を実現するメカニズムを意識した表現

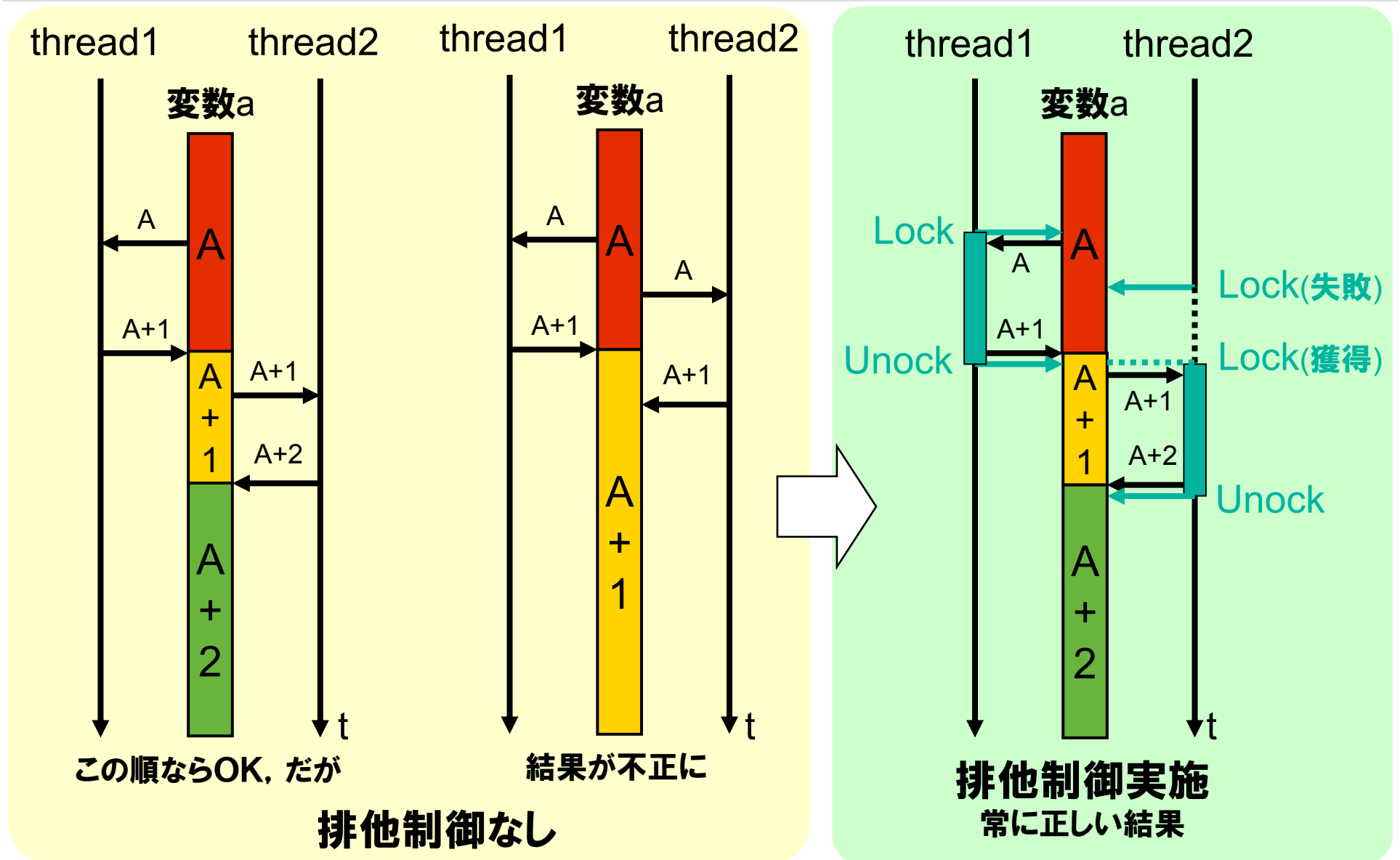
## Latch

- 実行時間の短いクリティカルセクションで利用  
Spin lockによってインプリメントされることが多い

cf. “この色で標記された部分” は、話者の解釈

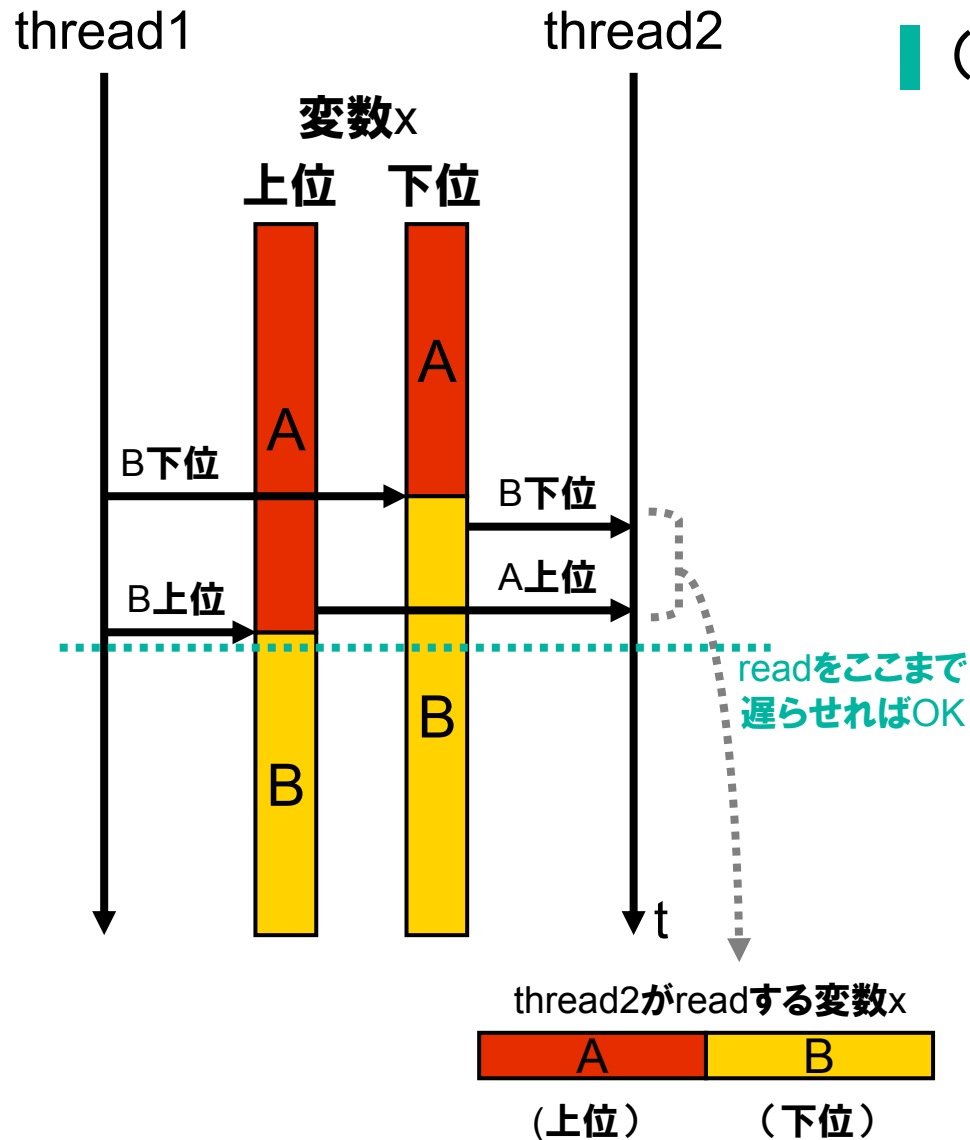
—|—|○ 本講演では、これらの違いを厳密に区別することなく進めさせていただきます。

# Lockが必要なのは..





# こんなパターンも



## (例) 32bit CPUで64bit dataを扱う場合

- C program

```
unsigned long long global_data;
```

```
void assign( unsigned long long arg )
{ global_data = arg; }
```

- 32bitのコンパイラで "cc -O"

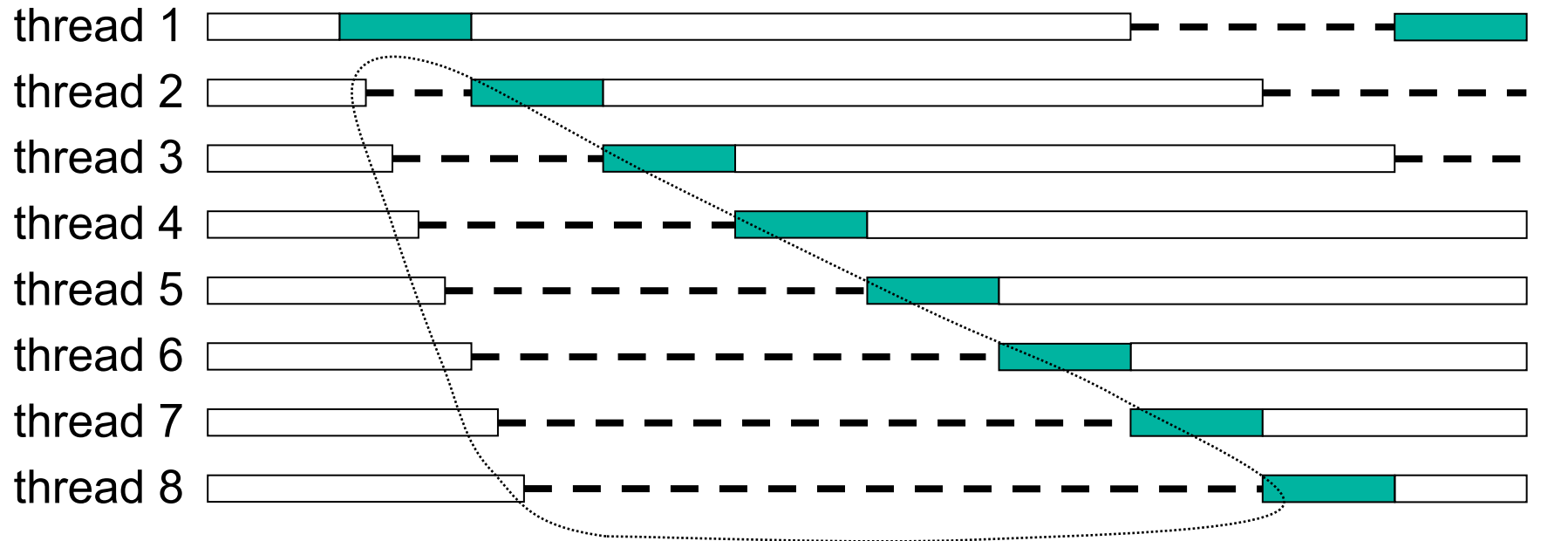
```
movl 8(%ebp), %eax      ← 下位
movl 12(%ebp), %edx     ← 上位
movl %eax, global_data ← 下位
movl %edx, global_data+4 ← 上位
```

- 64bitのコンパイラの場合

```
movq %rdi, global_data(%rip)
```


このメモリアクセスは  
不可分(atomic)

# ロック競合のボトルネック



処理が進まない  
⇒ 8CPU搭載していても、  
スループットは8倍にならない

→ t

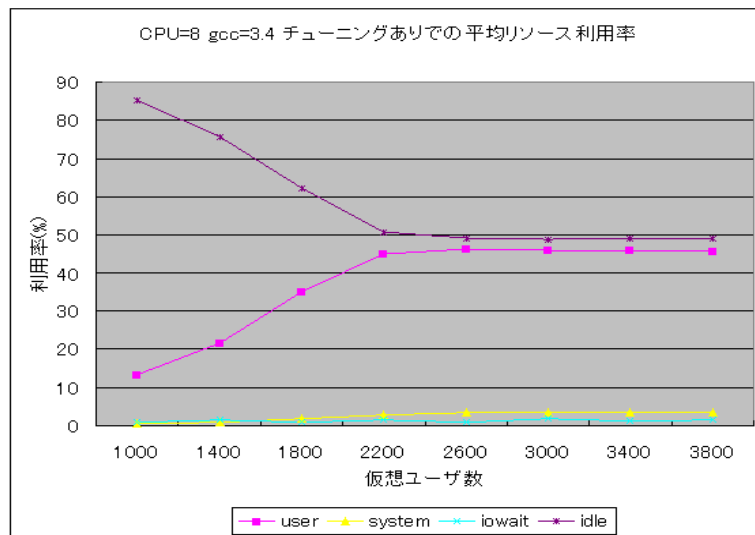
 排他実行区間  
(クリティカル・セクション)

# 実例 (DBT-1 on mysqlの測定結果)

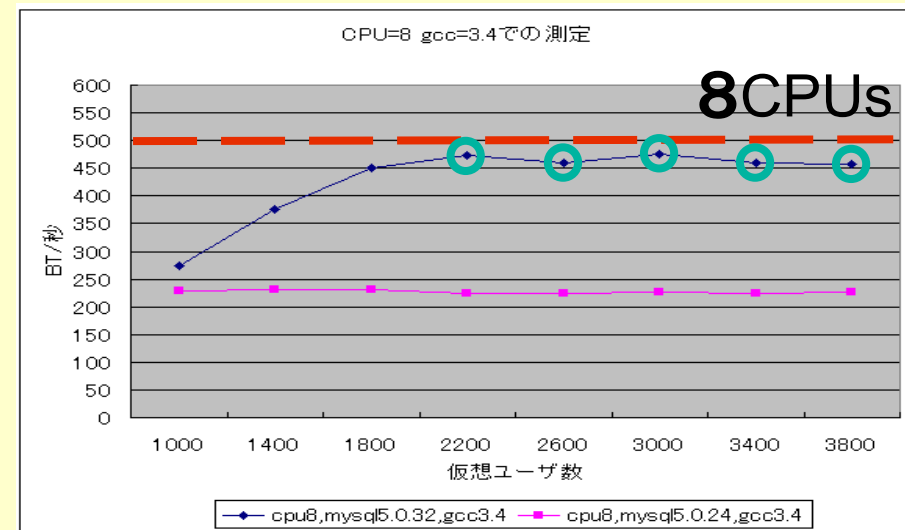
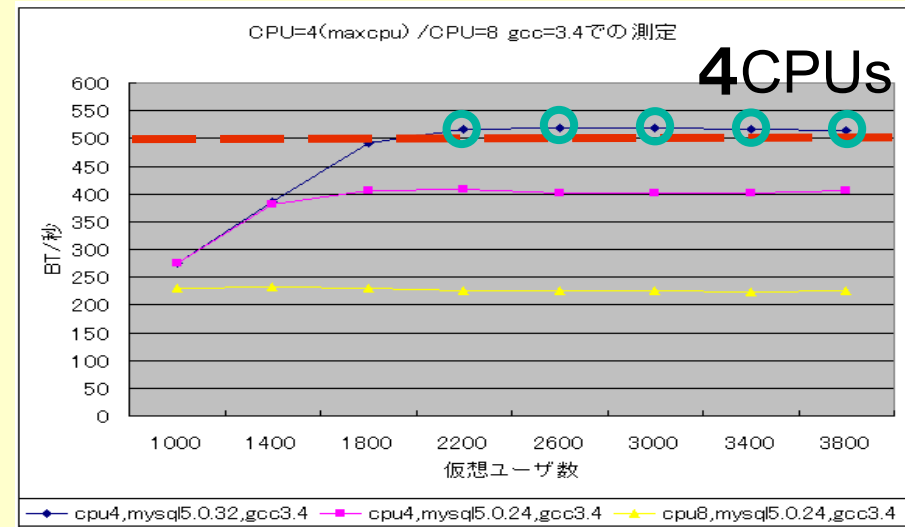
CPU強化(4→8)が性能向上に繋がっていない

- 4CPUと同程度
- CPU使用率は50%程度で飽和
- 他の物理資源(Disk, ネットワーク, メモリ)も飽和していない

CPU使用率(8CPU時) ↓



スループット →



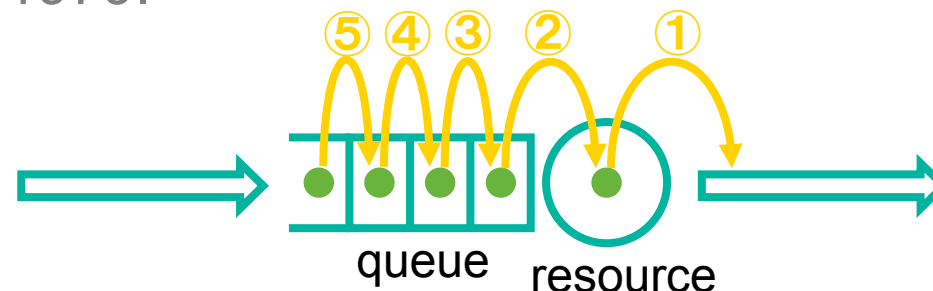
<http://ossipedia.ipa.go.jp/capacity/CS0612210243/>

<http://ossipedia.ipa.go.jp/capacity/EV0612260303>

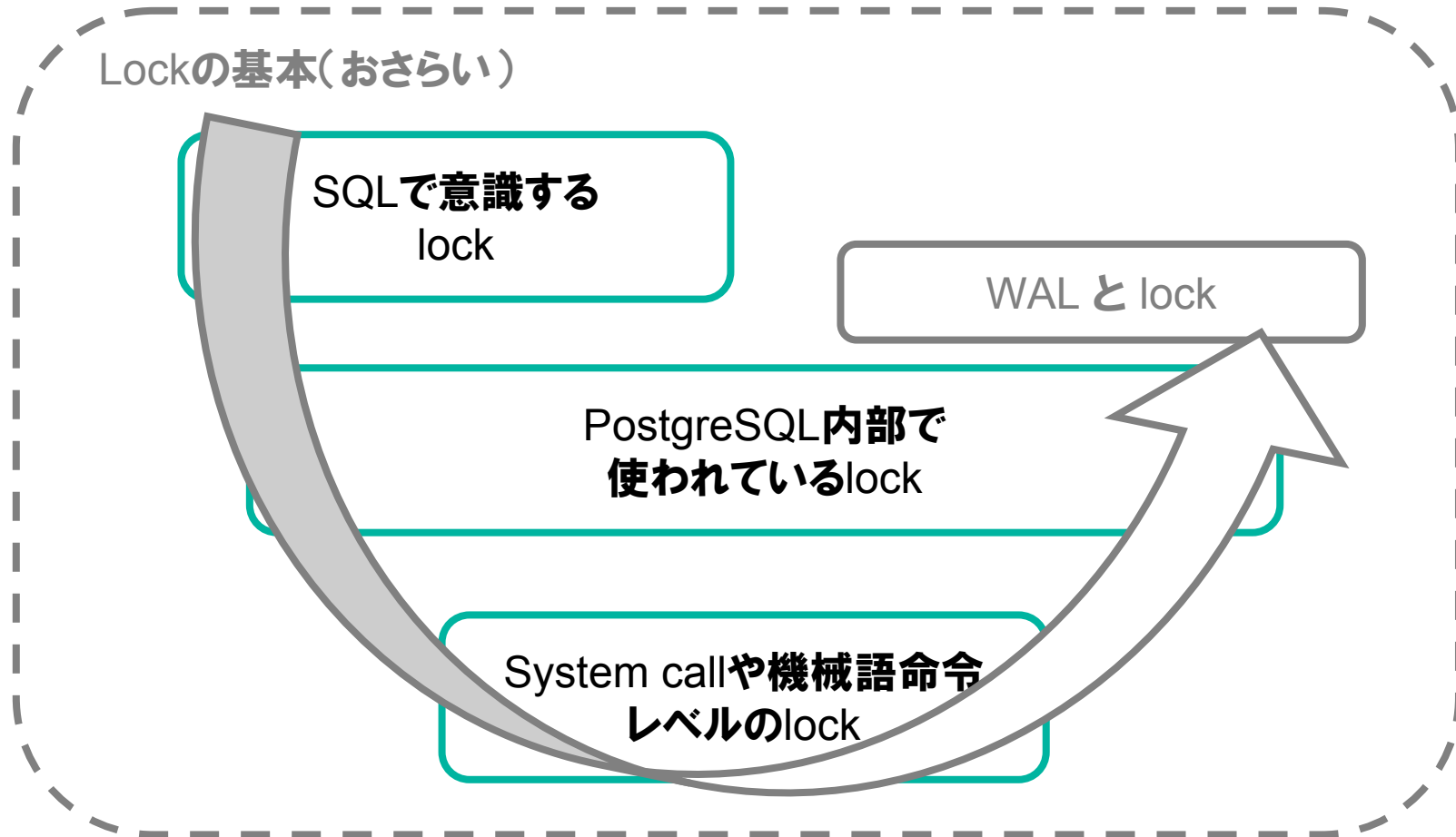
# 文献

Ryan Johnson , Ippokratis Pandis , Anastasia Ailamaki, [Critical sections: re-emerging scalability concerns for database storage engines](#), Proceedings of the 4th international workshop on Data management on new hardware, June 13-13, 2008, Vancouver, Canada.

Mike Blasgen , Jim Gray , Mike Mitoma , Tom Price, [The convoy phenomenon](#), ACM SIGOPS Operating Systems Review, v.13 n.2, p.20-25, April 1979.



# Agenda



# PostgreSQLマニュアル記載のLock (1)

## Table-level lock modes

**ACCESS SHARE** : SELECT

**ROW SHARE** : SELECT FOR UPDATE and SELECT FOR SHARE

**ROW EXCLUSIVE** : UPDATE, DELETE, and INSERT

**SHARE UPDATE EXCLUSIVE** :

VACUUM (without FULL), ANALYZE, and CREATE INDEX CONCURRENTLY

**SHARE** : CREATE INDEX (without CONCURRENTLY)

**SHARE ROW EXCLUSIVE** : none

**EXCLUSIVE** : none

**ACCESS EXCLUSIVE** :

ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER, and VACUUM FULL

Requested Lock Mode	Current Lock Mode							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

cf. <http://www.postgresql.org/docs/9.0/static/explicit-locking.html>

# PostgreSQLマニュアル記載のLock (2)

---

## Row-level locks

- Exclusive or shared locks
- Exclusive → Automatically acquired when the row is updated or deleted
- The lock is held until the transaction commits or rolls back
- Do not affect data querying; they block only writers to the same row
- “select the row with SELECT FOR UPDATE” **により, DBへの書込みなしに取得可能**

# Source codeとの対応(定義)

src/include/storage/lock.h

```
/* NoLock is not a lock mode, but a flag value meaning "don't get a lock" */
#define NoLock 0

#define AccessShareLock 1 /* SELECT */
#define RowShareLock 2 /* SELECT FOR UPDATE/FOR SHARE */
#define RowExclusiveLock 3 /* INSERT, UPDATE, DELETE */
#define ShareUpdateExclusiveLock 4 /* VACUUM (non-FULL),ANALYZE, CREATE
 * INDEX CONCURRENTLY */
#define ShareLock 5 /* CREATE INDEX (WITHOUT CONCURRENTLY) */
#define ShareRowExclusiveLock 6 /* like EXCLUSIVE MODE, but allows ROW
 * SHARE */
#define ExclusiveLock 7 /* blocks ROW SHARE/SELECT...FOR
 * UPDATE */
#define AccessExclusiveLock 8 /* ALTER TABLE, DROP TABLE, VACUUM
 * FULL, and unqualified LOCK TABLE */
```



# Source codeとの対応(操作)

src/backend/storage/lmgr/lmgr.c

```
void
LockRelationOid(Oid relid, LOCKMODE lockmode)
{
    LOCKTAG    tag;
    LockAcquireResult res;

    SetLocktagRelationOid(&tag, relid);

    res = LockAcquire(&tag, lockmode, false, false);
    ....
}

void
UnlockRelationOid(Oid relid, LOCKMODE lockmode)
{
    LOCKTAG    tag;

    SetLocktagRelationOid(&tag, relid);

    LockRelease(&tag, lockmode, false);
}
```

*/\*  
\* LockRelationOid  
\*  
\* Lock a relation given only its OID. This should generally be used  
\* before attempting to open the relation's relcache entry.  
\*/*

*/\*  
\* UnlockRelationOid  
\*  
\* Unlock, given only a relation Oid. Use UnlockRelationId if you can.  
\*/*

cf. transaction終了時のlock解放はLockReleaseAll()により行なわれます

## Table-level locks

- LockRelationOid(), ConditionalLockRelationOid(), UnlockRelationId(), UnlockRelationOid(), LockRelation(), ConditionalLockRelation(), UnlockRelation(), LockRelationIdForSession(), UnlockRelationIdForSession(), LockRelationForExtension(), UnlockRelationForExtension()

## Page-level locks

- LockPage(), ConditionalLockPage(), UnlockPage()

## Row-level locks

- LockTuple(), ConditionalLockTuple(), UnlockTuple()

(以下, 略)

- \* Obtain a tuple-level lock. This is used in a less-than-intuitive fashion
- \* because we can't afford to keep a separate lock in shared memory for every
- \* tuple. See heap\_lock\_tuple before using this!

cf. LockTuple()のコメント(source codeから引用)にある通り, 総てのlock取得がlmgr.cにある関数を使って行なわれる(=後出のhash管理される)わけではなく, 実行効率を上げる工夫が施されている場合があります.

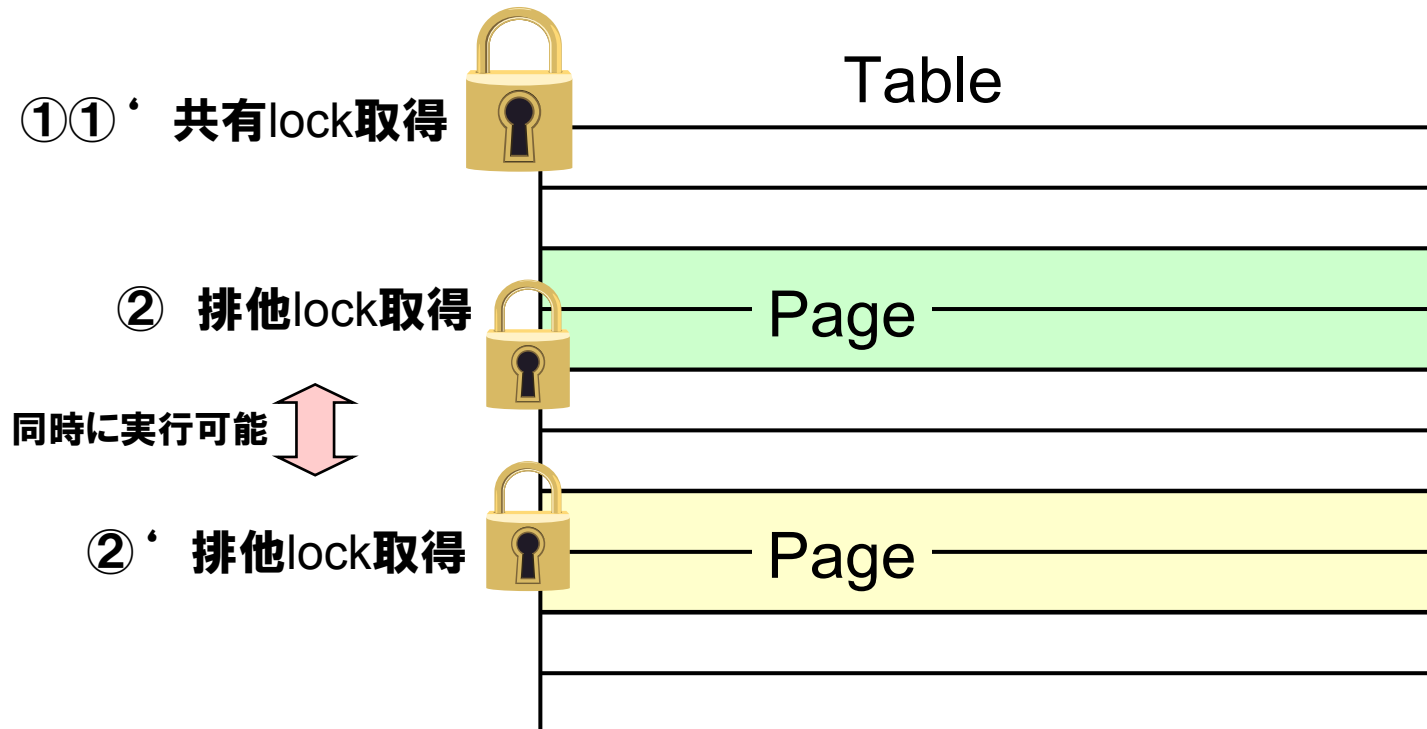
# 階層lock（目的, 使われ方）

## 上位(粒度の粗い)lockを共有モードで取得

- 排他モードでの上位lock取得を要する稀な処理への備え

## 下位(粒度の細かい)lockは排他モードで取得

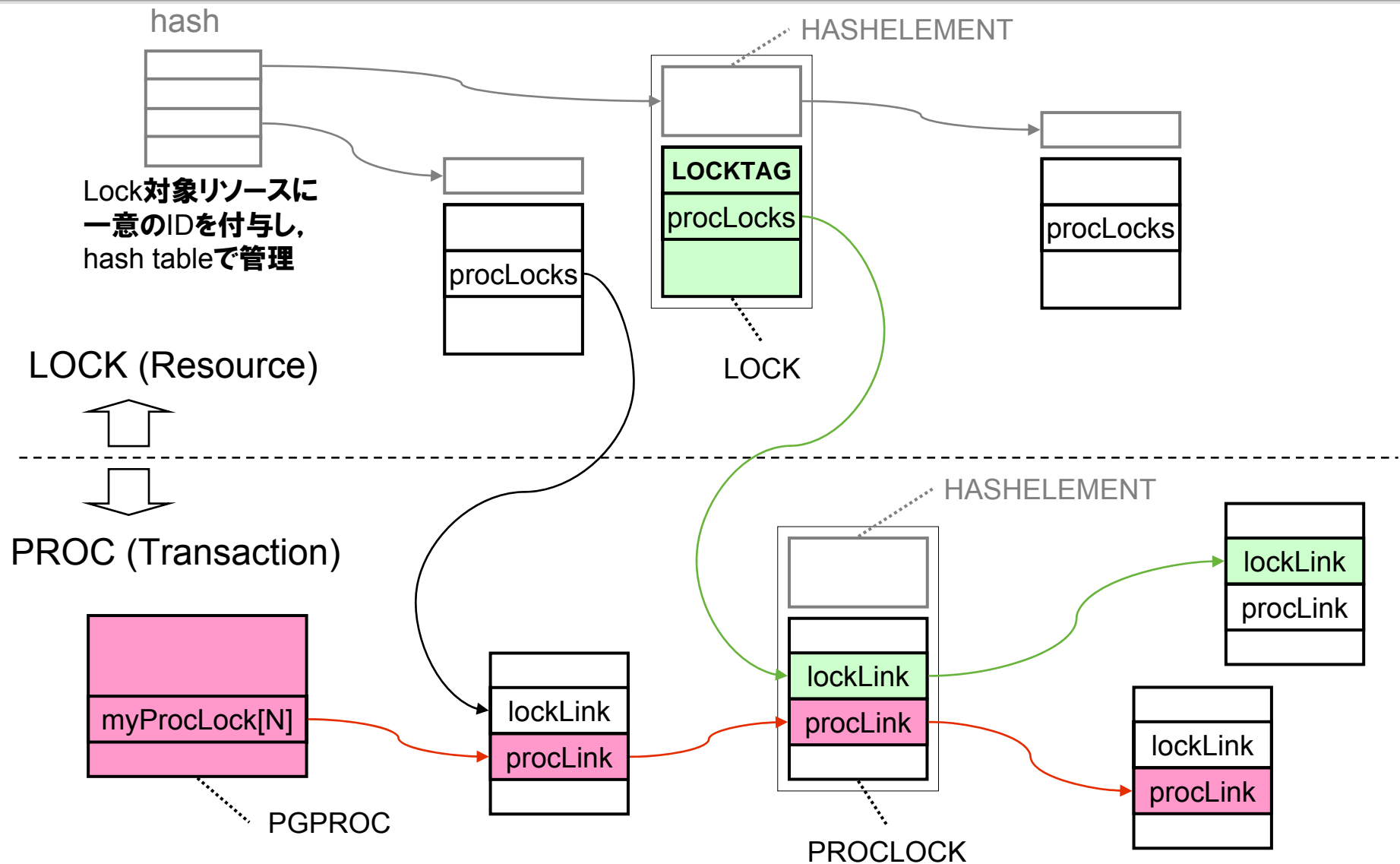
- 下位lockの対象が異なる処理は, 同時に実行可 → 並列度を高くできる



cf. 本図は, 階層lockの概念を簡易に説明するためのもので, PostgreSQLの実装内容を直接説明するものではありません.

# Lockのインプリメント

src/backend/storage/lmgr/lock.c, src/include/storage/lock.h  
 src/backend/utils/hash/dynahash.c, src/include/utils/hsearch.h



cf. LOCALLOCKは省略

# LOCK, PROCLOCKの挿入・削除

## 挿入 – LockAcquire()

- LOCKエントリがhashになれば、エントリを作成してhashに登録
- PROCLOCKエントリがhashになれば、エントリを作成してhashに登録
- LOCKエントリのprocLocksにPROCLOCKエントリを繋ぐ
- PGPROCエントリのmyProcLockにPROCLOCKエントリを繋ぐ

## 削除 – LockRelease(), LockReleaseAll()

- PGPROCエントリのmyProcLockからPROCLOCKエントリを外す
- LOCKエントリのprocLocksからPROCLOCKエントリを外す
- PROCLOCKエントリが管理するLockがなければ、hashから削除し、エントリも解放する
- LOCKエントリを使うPROCLOCKエントリがなければ、hashから削除し、エントリも解放する

**挿入・削除は、上記操作を不可分(atomic)に行う必要がある  
→ エントリをパーティションに分け、lwlockにより排他制御**

# LWLock (Lightweight lock manager) データ構造

PostgreSQLにおいて、最も多用されているLock [src/backend/storage/lmgr/lwlock.c](https://github.com/postgres/postgres/blob/master/src/backend/storage/lmgr/lwlock.c)

- 下記の内部データ構造を使用

```
typedef struct LWLock
{
    slock_t    mutex;
    bool       releaseOK;
    char       exclusive;
    int        shared;
    PGPROC     *head;
    PGPROC     *tail;
    /* tail is undefined when head is NULL */
} LWLock; slock_t
```

**atomicに変更する必要あり**

/\* Protects LWLock and queue of PGPROCs \*/  
/\* T if ok to release waiters \*/  
/\* # of exclusive holders (0 or 1) \*/  
/\* # of shared holders (0..MaxBackends) \*/  
/\* head of list of waiting PGPROCs \*/  
/\* tail of list of waiting PGPROCs \*/

# LWLock (Lightweight lock manager) 動作

## LWLockメンバのatomicな変更 → mutexによる排他制御

- 基本的には spin lock `src/backend/storage/lmgr/s_lock.c, src/include/storage/s_lock.h`

- `test_and_set (= lock; xchg命令)` によりmutex獲得を試みる

何回か失敗すれば`pg_usleep()`する

- mutex解放イベントが通知(待っているプロセスがwake up)されるわけではない

この動作は、ユーザ空間だけではインプリメント不可  
カーネルのサポートが必要

- `pg_usleep()`を何回か行なうと, `eelog(PANIC, ... );` する

cf. `slock()` @ `src/backend/storage/lmgr/s_lock.c` がそのようにコーディングされている, という意味  
具体的には, `if (++delays > NUM_DELAYS) s_lock_stuck(lock, file, line);` でcallされる  
`s_lock_stuck()`内で, `eelog(PANIC, ...)`が実行される.

## LWLockが獲得できない場合

- headとtailで管理されるlistに自プロセスを繋ぐ
- 自プロセスに対応付けられているsemaphoreで待つ(sleepする)

# LWLock 用途

## LWLockIdの割り当て

src/include/storage/lwlock.h

```
typedef enum LWLockId
{
    BufFreelistLock,          ShmemIndexLock,          OidGenLock,          XidGenLock,
    ProcArrayLock,          SInvalReadLock,         SInvalWriteLock,    WALInsertLock,
    WALWriteLock,          ControlFileLock,        CheckpointLock,     CLogControlLock,
    SubtransControlLock,    MultiXactGenLock,       MultiXactOffsetControlLock, MultiXactMemberControlLock,
    RelCacheInitLock,      BgWriterCommLock,      TwoPhaseStateLock,  TablespaceCreateLock,
    BtreeVacuumLock,       AddinShmemInitLock,    AutovacuumLock,     AutovacuumScheduleLock,
    SyncScanLock,          RelationMappingLock,   AsyncCtlLock,       AsyncQueueLock,
    /* Individual lock IDs end here */

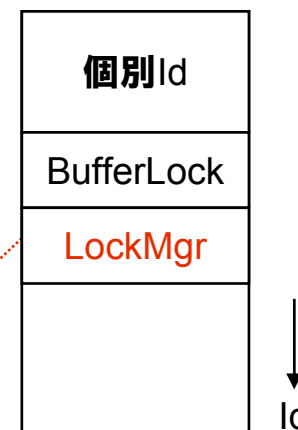
    FirstBufMappingLock,
    FirstLockMgrLock = FirstBufMappingLock + NUM_BUFFER_PARTITIONS,

    /* must be last except for MaxDynamicLWLock: */
    NumFixedLWLocks = FirstLockMgrLock + NUM_LOCK_PARTITIONS,
                                     (Default値は16)

    MaxDynamicLWLock = 1000000000
} LWLockId;
```

WAL関連のLWLockId

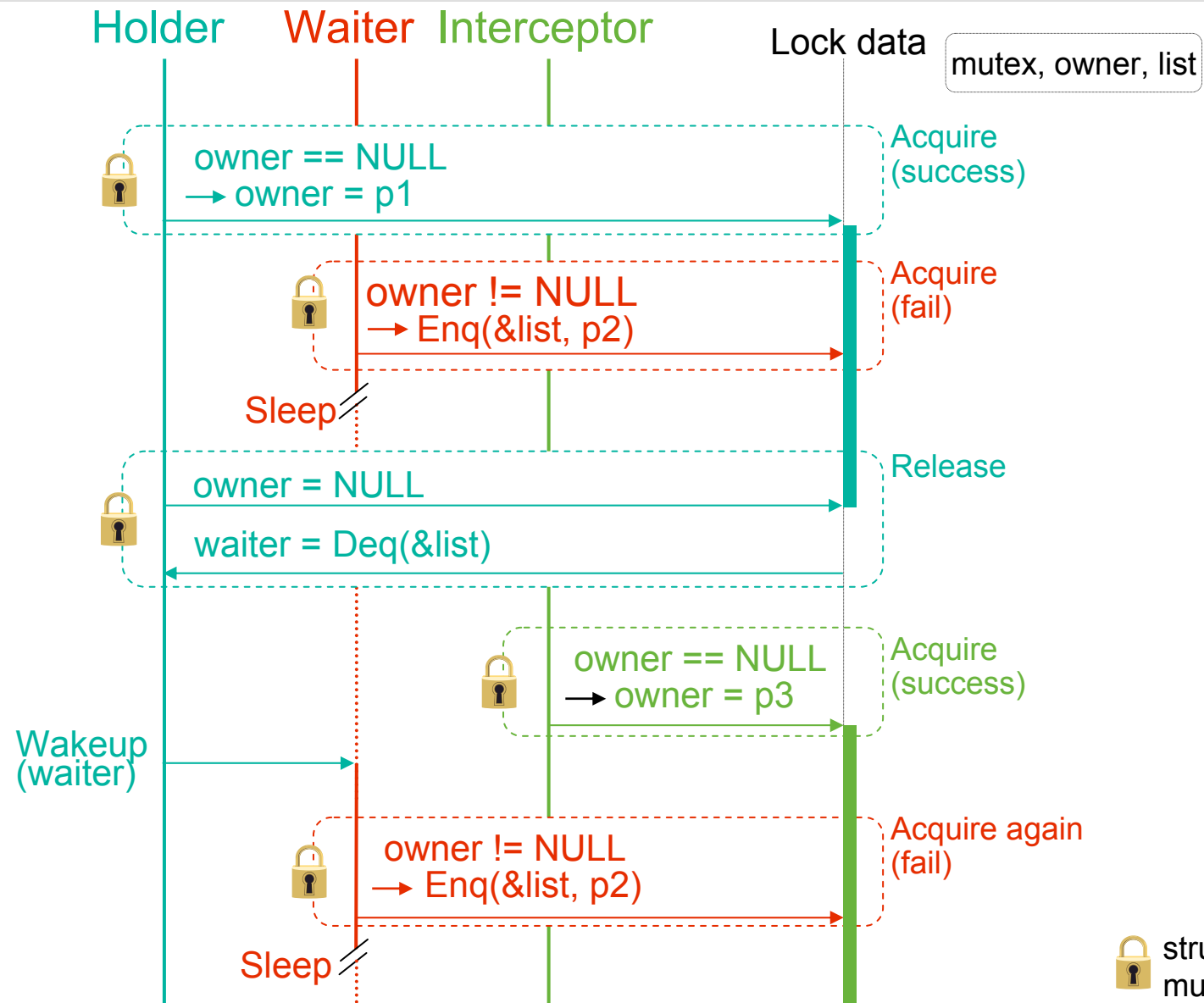
実際に割り当てられる数は NumLWLocks() のreturn値



LOCK, PROCLOCKをパーティションに分けて排他制御する際のLWLockId

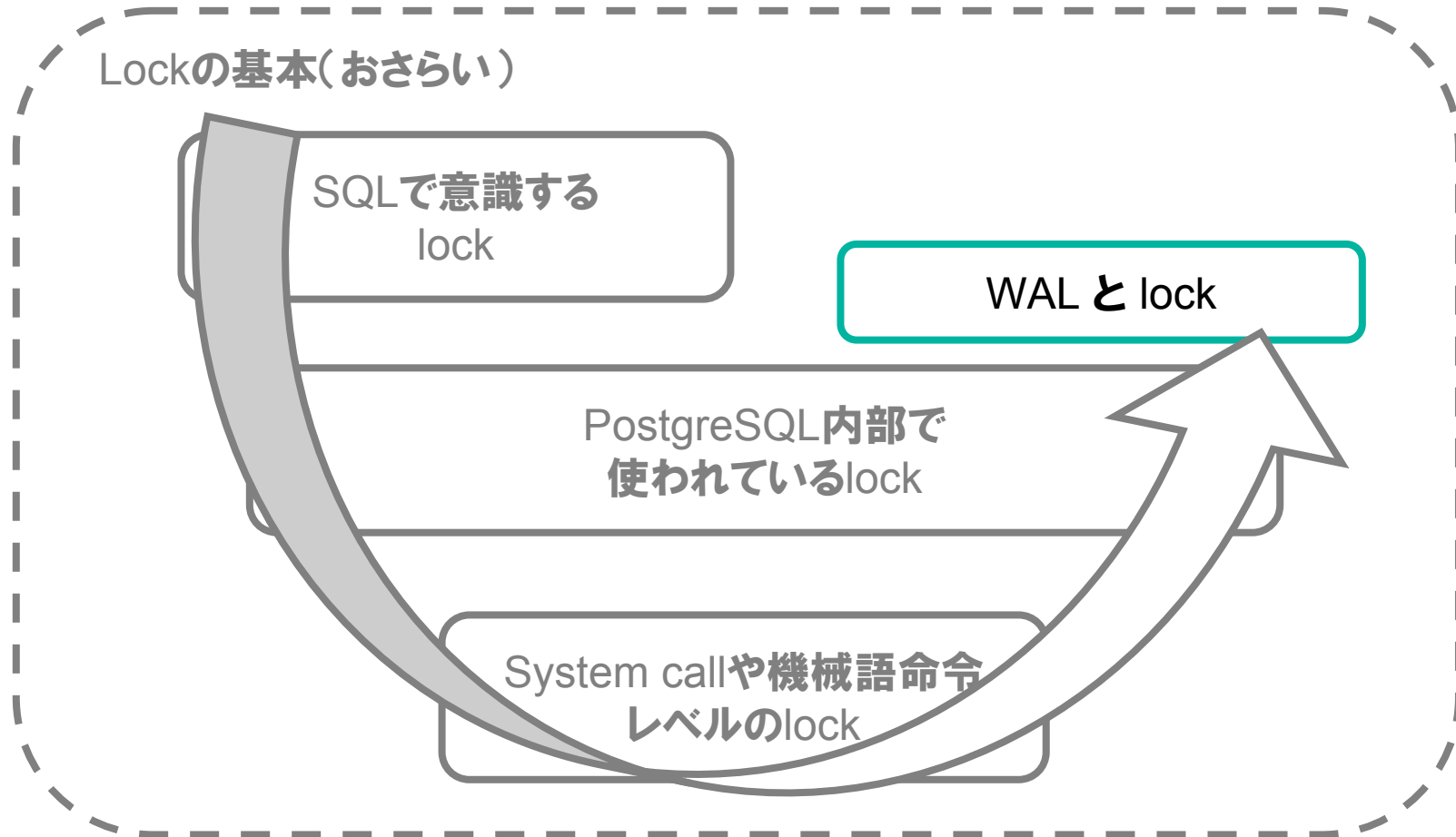


# LWLockの横取り 問題となる可能性は??



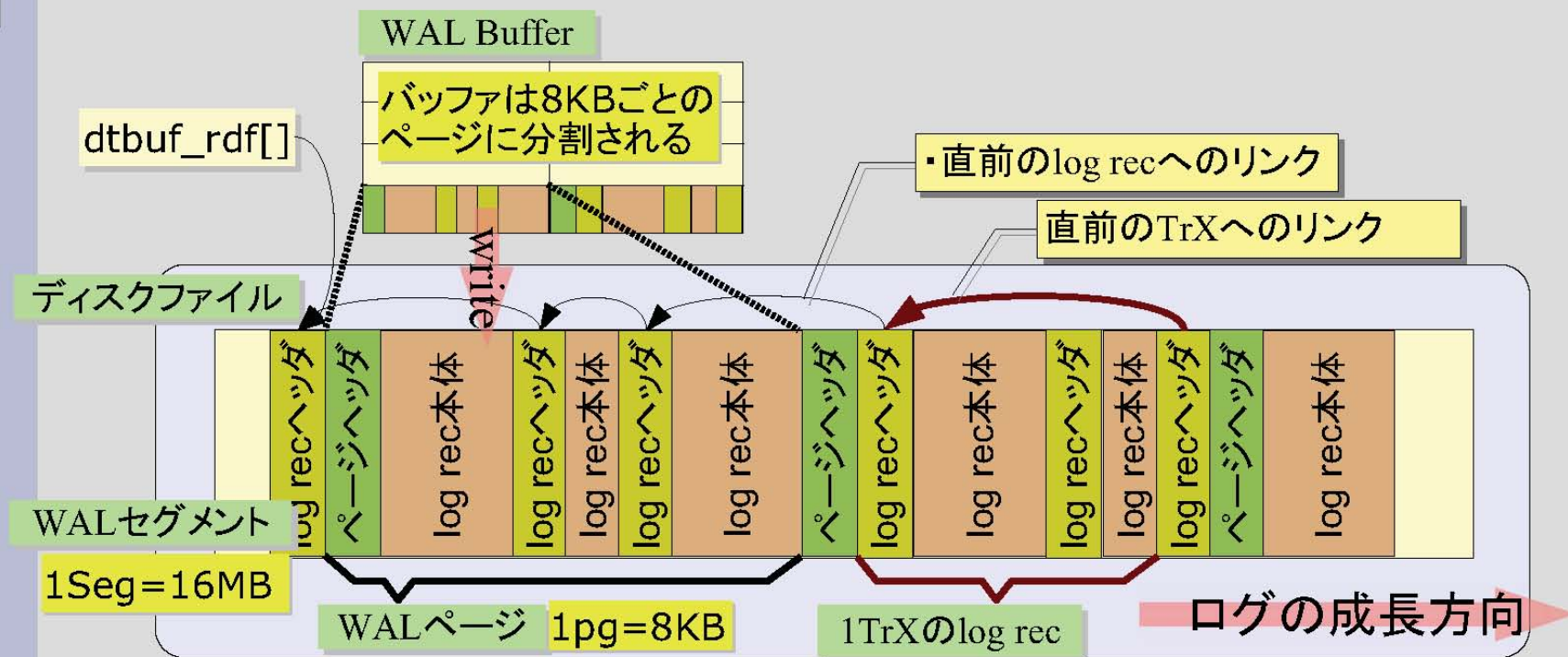
struct LWLock 内の mutexによる排他制御

# Agenda



# Loggingシステムのデータ構造

- XLogInsert関連の主要データ(その2)
  - ディスク上のログデータの配置



# WAL雑学

---

## Postgresの初期は「WALなし」が特徴

- Michael Stonebraker, [The Design of the POSTGRES Storage System](#), Proceedings of the 13th International Conference on Very Large Data Bases, pp.289-300, 1987.  
... First, the storage manager supports transaction management but does so **without using a conventional write ahead log (WAL)**. In fact, there is no code to run at recovery time, and consequently recovery from crashes is essentially instantaneous.

## PostgreSQL 7.1で追加された機能 (<http://ja.wikipedia.org/wiki/PostgreSQL>)

### 「WALなし」の再検討

- 川田明良, 藤田悦郎, 日高東潮, 高速I/O環境でのOLTP負荷に対する PostgreSQLのCPUスケーラビリティに関する一考察, DEIM Forum 2011 D5-2.  
... CPUスケーラビリティの障害要因はジャーナル(WAL)編集での**バッファ挿入の直列化**(cf. **WALInsertLock**と思われる)であった 本論文では回避方法としてPostgreSQLのWAL導入以前のWrite-Through方式を採用し, ...  
SSDにより, write-throughのコストは低下しているとの想定

# WALによる性能低下を緩和する方策

■ **非同期commit** (石井達夫, <http://journal.mycom.co.jp/special/2007/postgresql/007.html>, 2007/11/20)

- **期待度大のバージョンアップ - PostgreSQL 8.3の改良点を徹底分析**

8 **非同期コミット(2) - トランザクションの消失を最小限に**

**非同期コミットのメリット**

これ(cf. fsync=off設定)に対して、8.3で追加された非同期コミットでは、こうしたリスクを避けることができる。データベースの整合性が失われることはなく、**クラッシュ時にも直近のトランザクションが失われるだけである**。不運にも同期書き込み前にシステムがクラッシュするなどしてトランザクションが失われた場合は、それを再実行すれば良い。

→ **直近とはいえ、commitした(はずの)トランザクションが失われて良いのか？**

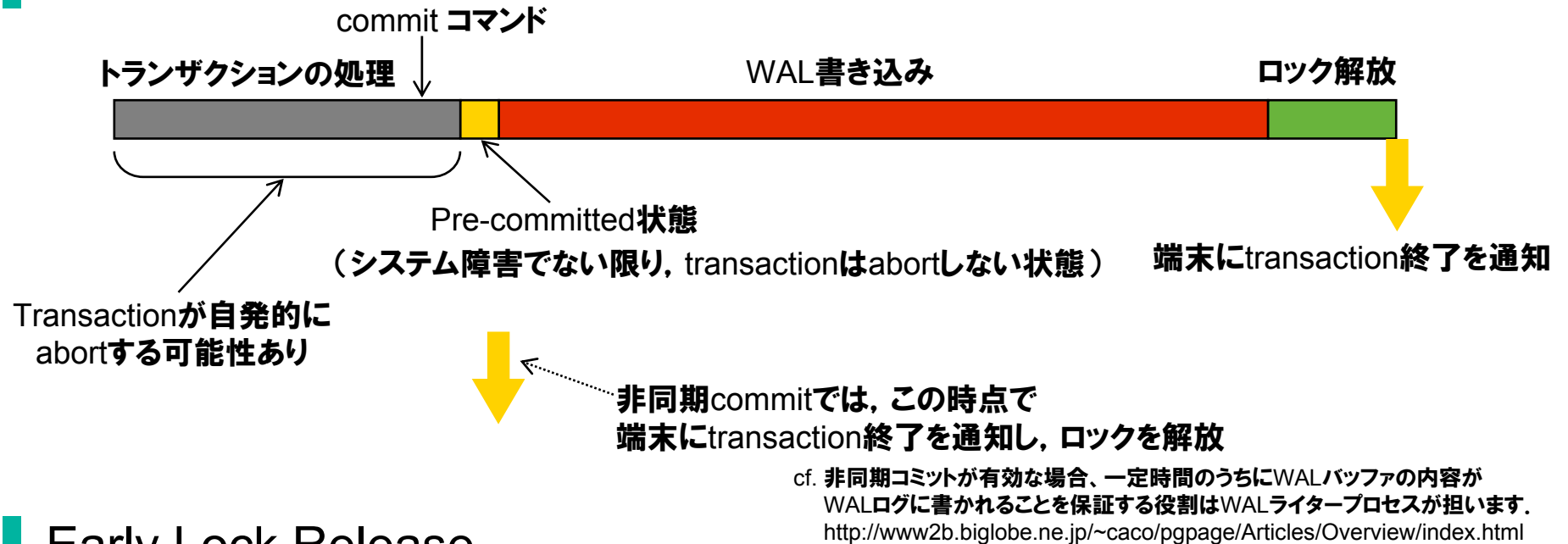
■ **Early Lock Release**

- Johnson, R. and Pandis, I. and Stoica, R. and Athanassoulis, M. and Ailamaki, A., [Aether: A scalable approach to logging](#), Proceedings of the VLDB Endowment VLDB Endowment, Volume 3 Issue 1-2, September 2010.

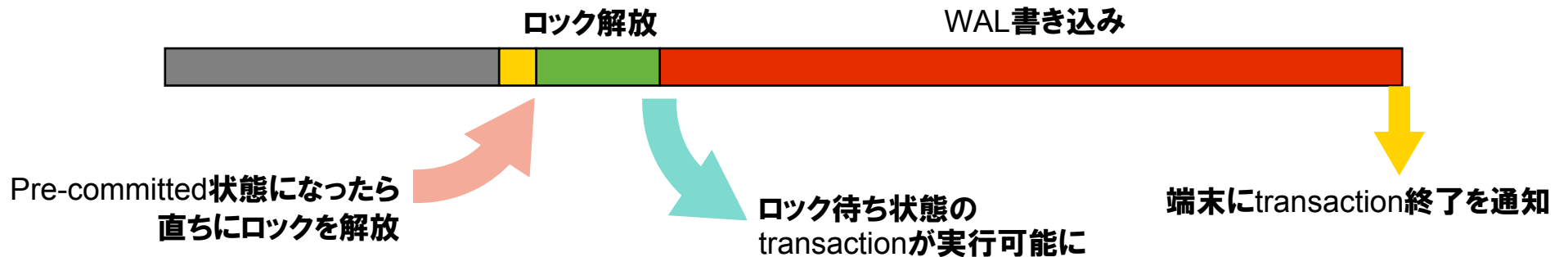
Early Lock Release (ELR) removes log flush latency from the critical path by ensuring that only the committing transaction must wait for its commit operation to complete: ... We hypothesize that this is largely due to the effectiveness of **asynchronous commit** ...

# Early Lock Release (イメージ)

## 従来

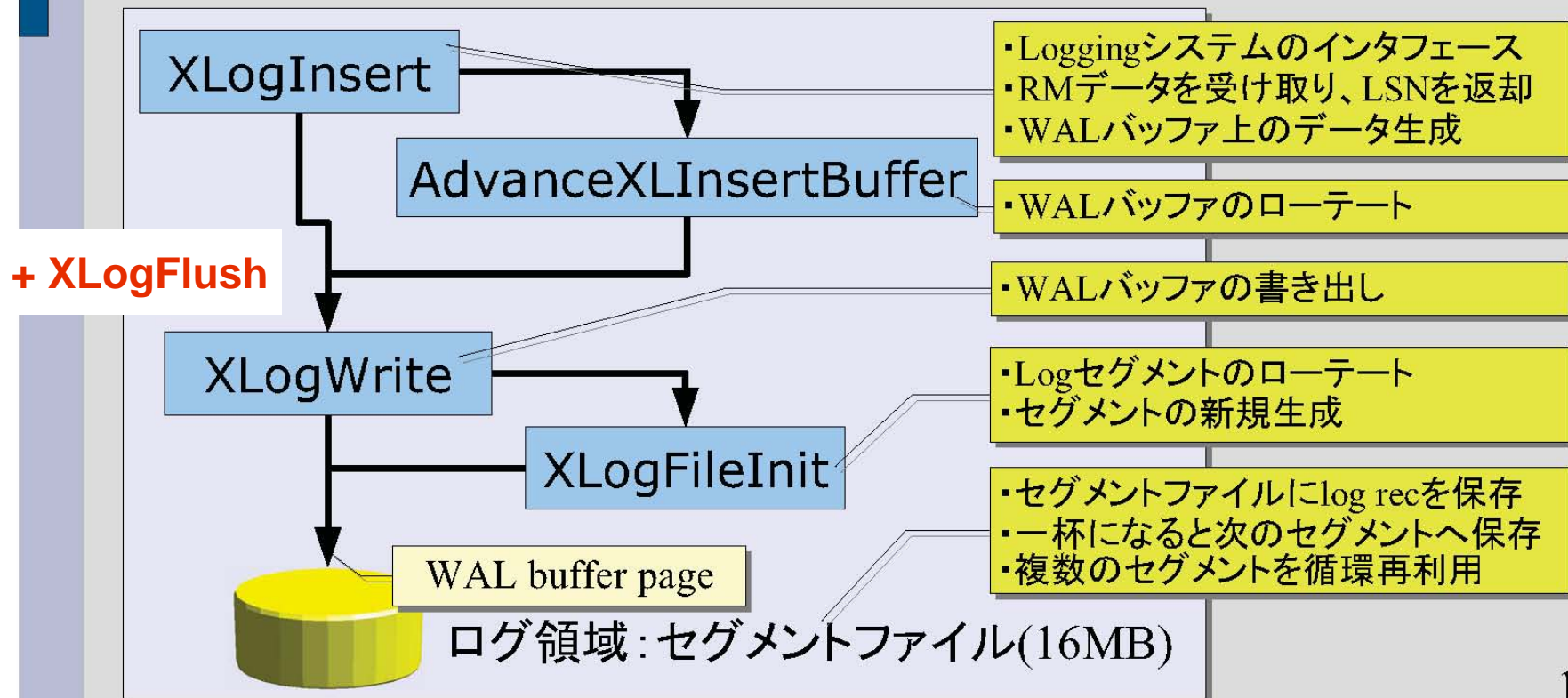


## Early Lock Release



# Loggingシステム内部構造

- 関数レベルでの内部構造(下図)
  - 在り処 `src/backend/access/transam/xlog.c`



# Commit に伴う WAL書き込み動作

## Stack trace

(gdb) bt

```
#0 pg_fdatasync (fd=36) at fd.c:299
#1 0x080c8bcf in issue_xlog_fsync () at xlog.c:6392
#2 0x080beeb4 in XLogWrite (WriteRqst={Write = {xlogid = 0, xrecoff = 6281584}, Flush = {xlogid = 0, xrecoff = 6281584}}, flexible=0 '¥0', xlog_switch=0 '¥0') at xlog.c:1614
#3 0x080bf2e9 in XLogFlush (record={xlogid = 0, xrecoff = 6281584}) at xlog.c:1741
#4 0x080b8a1c in RecordTransactionCommit () at xact.c:949
#5 0x080b93dd in CommitTransaction () at xact.c:1675
#6 0x080b9d3a in CommitTransactionCommand () at xact.c:2274
#7 0x0824b647 in finish_xact_command () at postgres.c:2322
#8 0x082493df in exec_simple_query (query_string=0x84ac870 "create table abc(a int);") at postgres.c:1017
#9 0x0824d15f in PostgresMain (argc=4, argv=0x8454520, user_name="postgres") at postgres.c:3572
#10 0x08217982 in BackendRun (port=0x8467d18) at postmaster.c:3207
#11 0x08216f0a in BackendStartup (port=0x8467d18) at postmaster.c:2830
#12 0x08214928 in ServerLoop () at postmaster.c:1274
#13 0x08214335 in PostmasterMain (argc=1, argv=0x8451578) at postmaster.c:1029
#14 0x081b6707 in main (argc=1, argv=0x8451578) at main.c:188
```

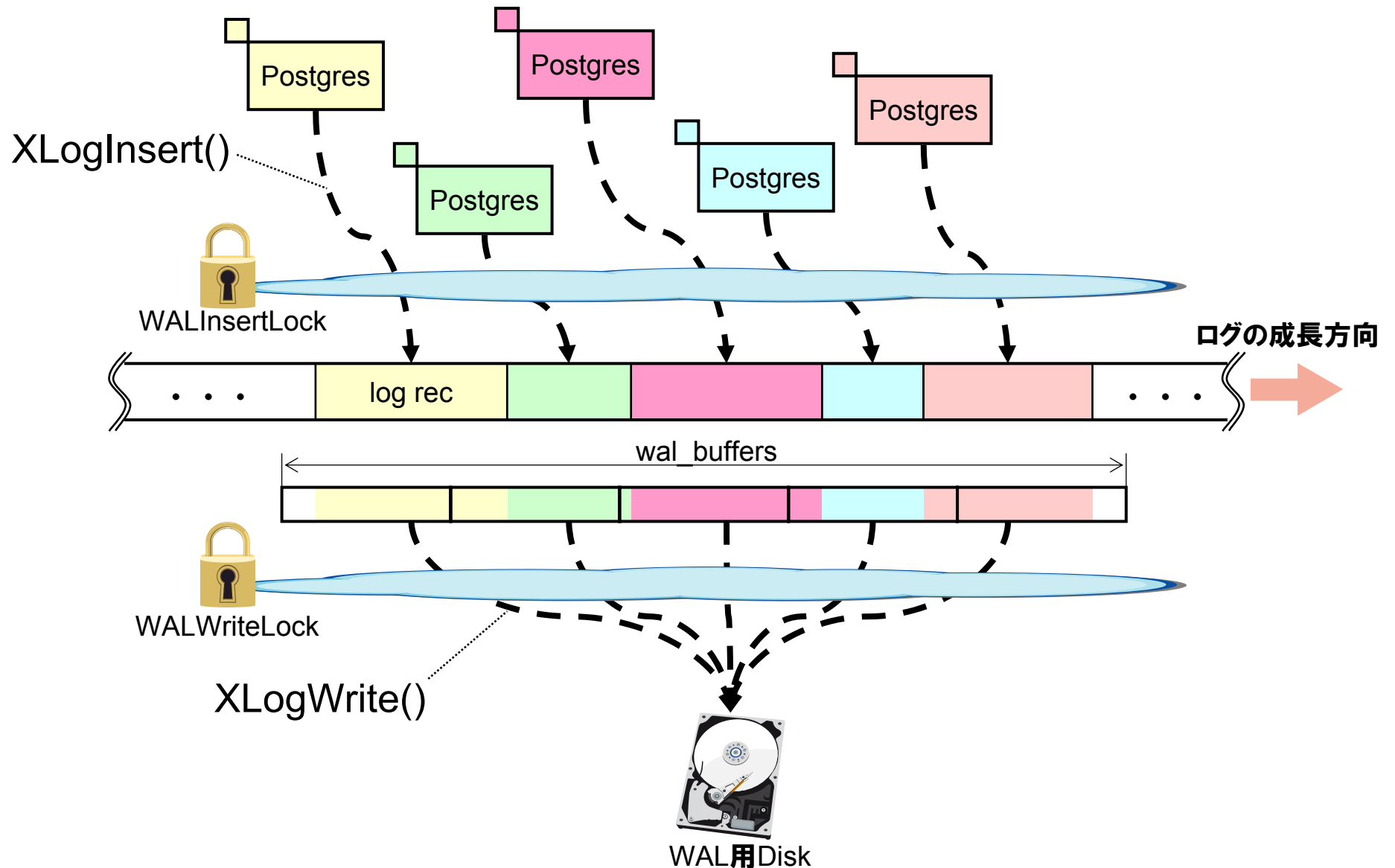
XLogFlushの直前で  
XLogInsertを実行

トランザクション処理本体

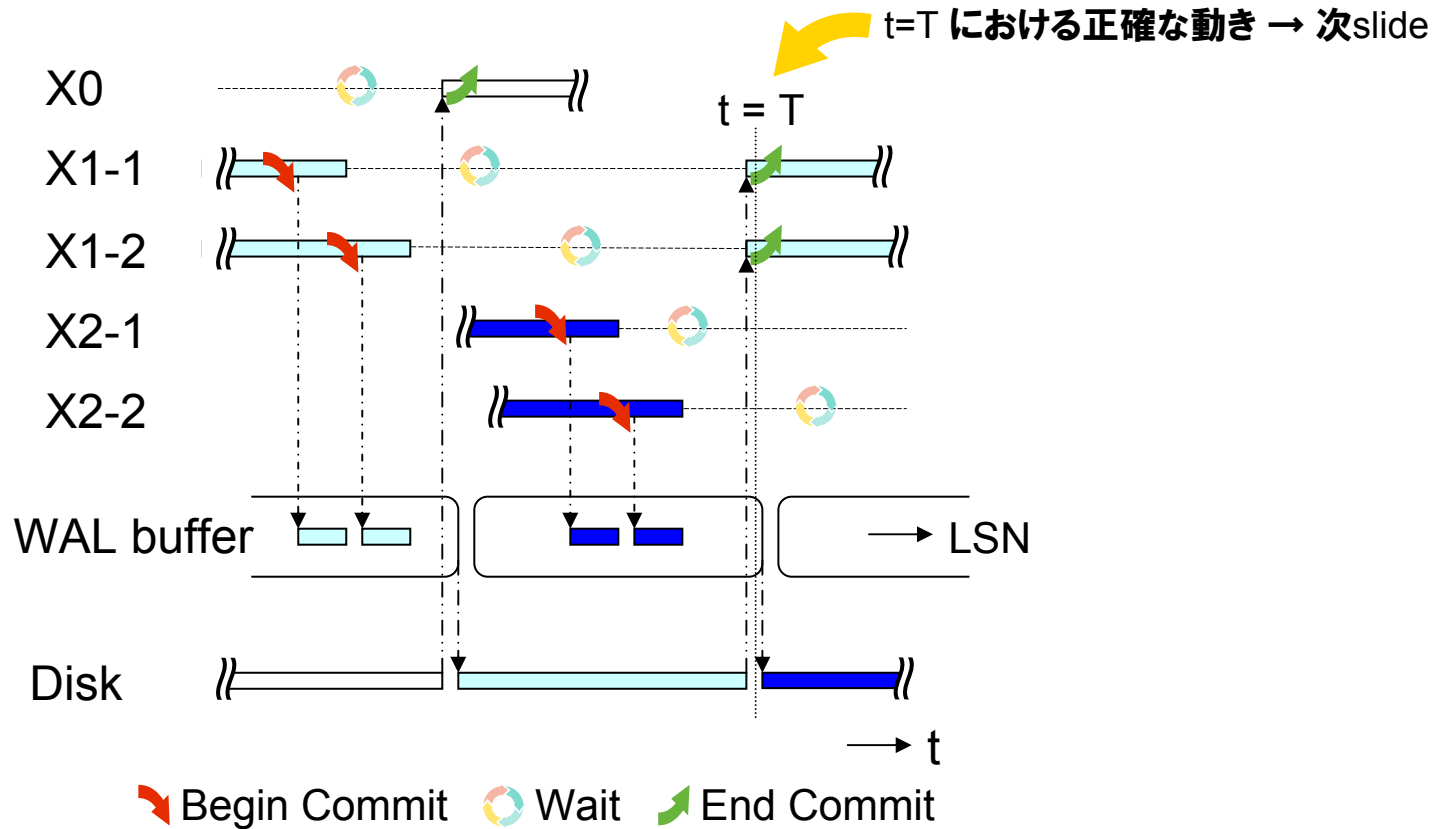
<http://d.hatena.ne.jp/taedium/20090215/p1>



# WAL と LWLock

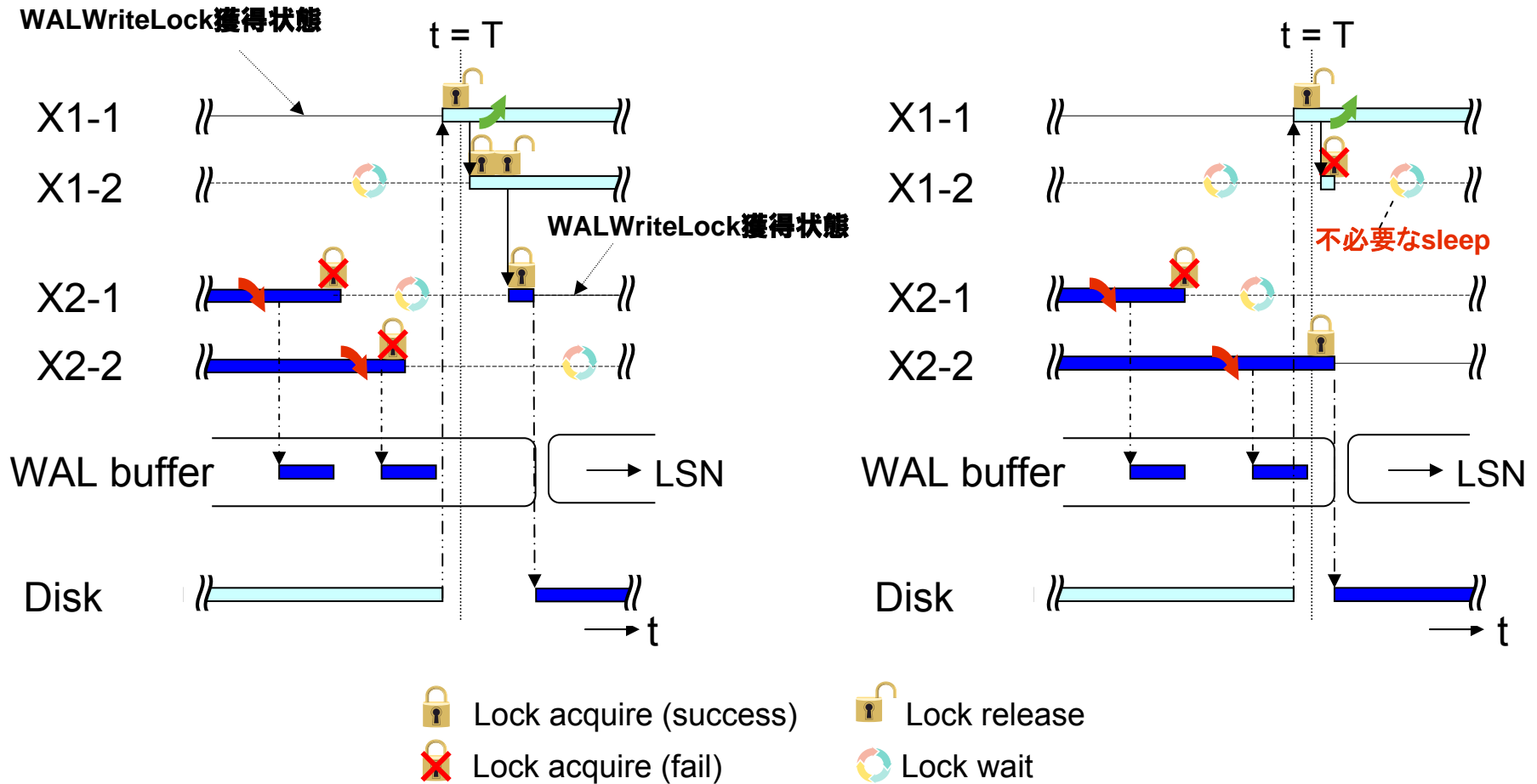


# Group commit



複数トランザクションのlogを1回のdisk accessでwriteする

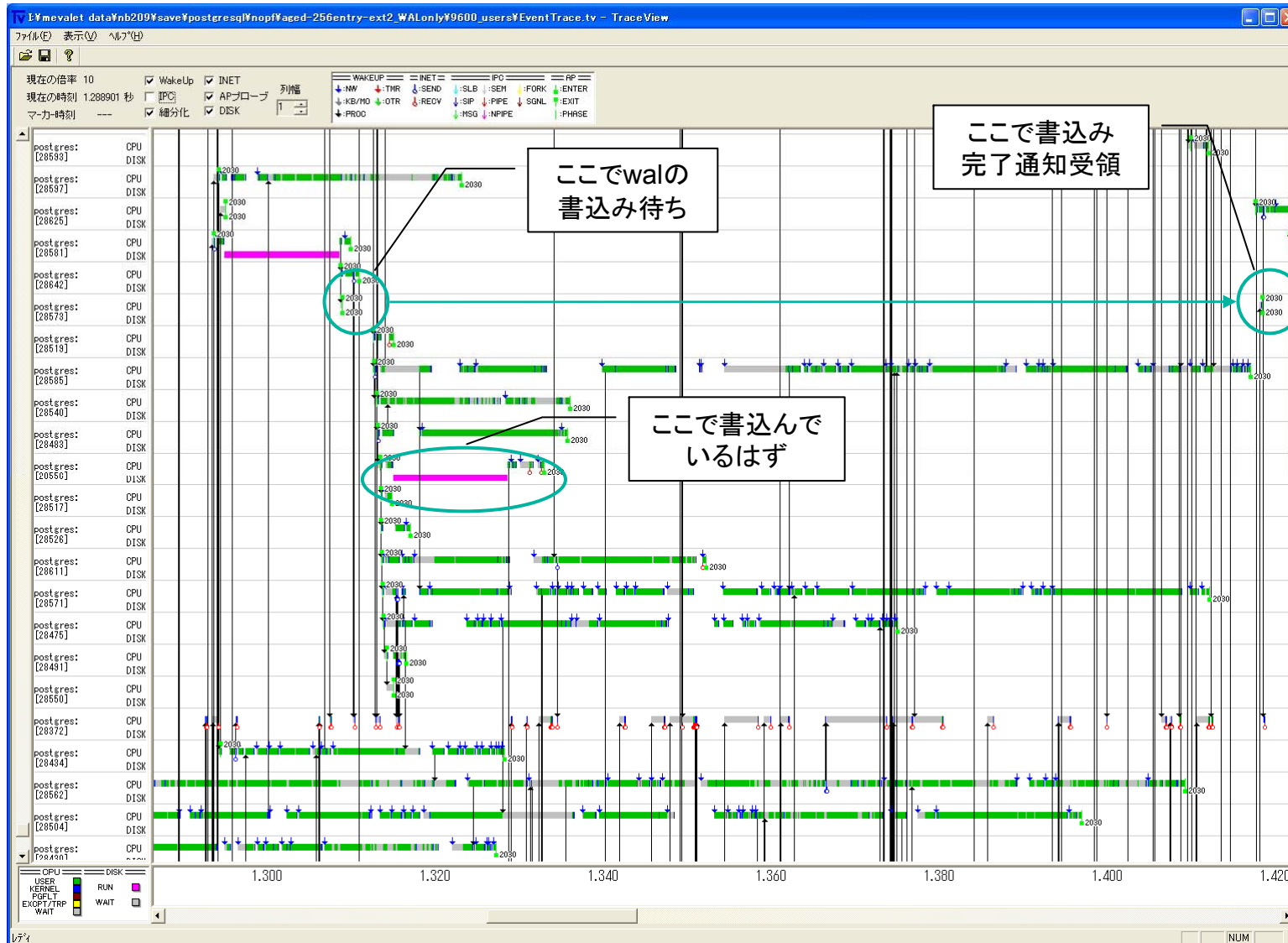
# 複数のWAL書き込み待ちプロセスをwake up



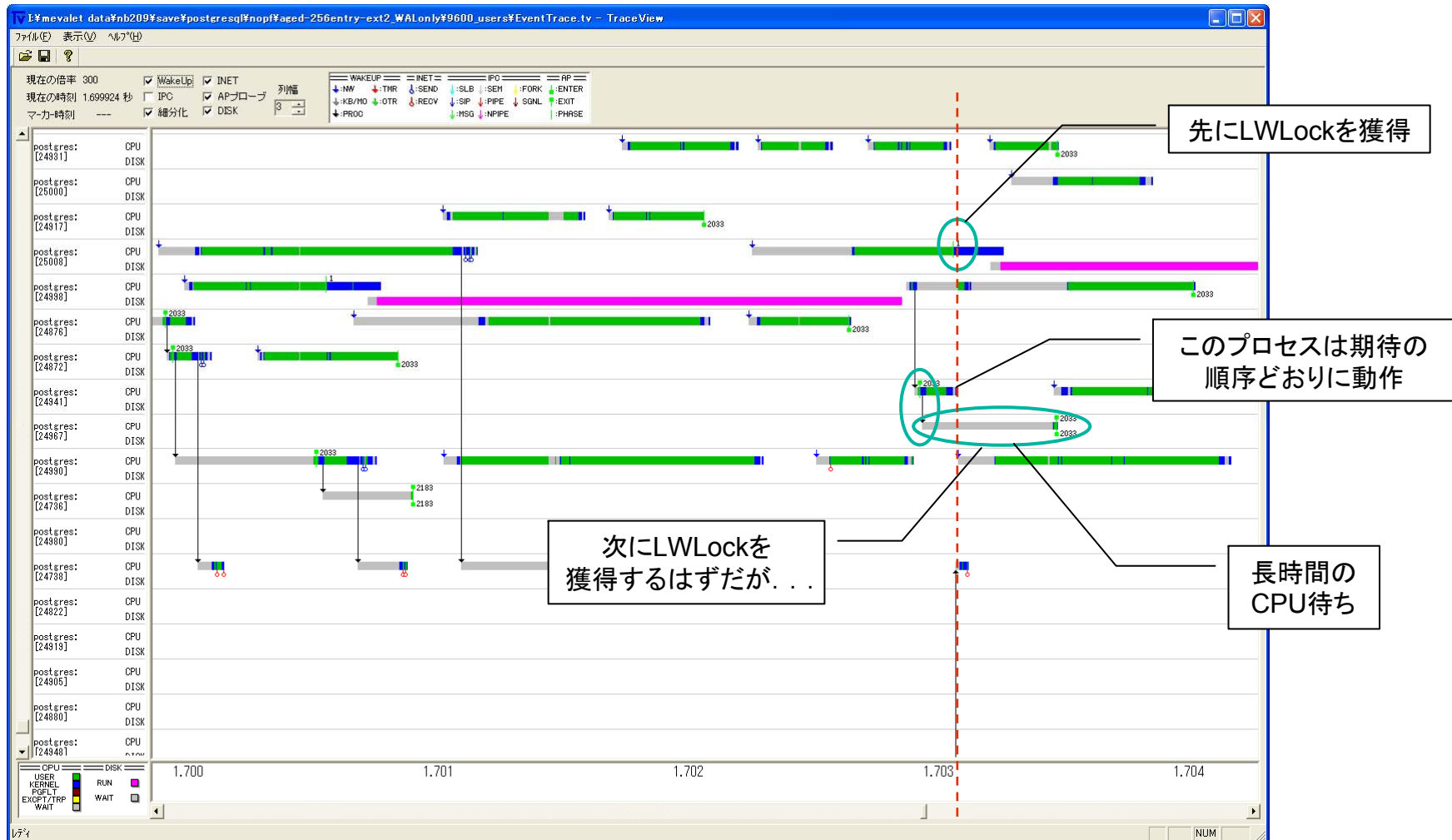
期待どおりの順番でwake up

Wake upの順番が狂うと...

# LWLock横取り 実例 (with mevalet™)



# LWLock横取り 拡大図 (with mevalet™)



300倍

# まとめ

---

## Lock

- SQLレベルで意識するLock, 内部で使用しているLWLockを中心に, それらの基本とインプリメントを source code も参照して説明

## WAL : 今後の評価に期待

- Early Lock Release
  - 非同期commit と同程度の効果がある？
- LWLock横取り
  - 影響の程度は？
  - 対策は？
- その他一般
  - SSDで性能向上するか？
  - Write Through復活は意味あるか？

## (参) 話者の関連発表

---

堀川 隆, データベースにおけるマルチプロセッサスケールビリティボトルネックの分析手法, 情報処理学会論文誌, 第51巻, 第7号, 2010年.

T. Horikawa and A. Fukuda, A method for analysis and solution of scalability bottleneck in DBMS, SoICT '10 Proceedings of the 2010 Symposium on Information and Communication Technology, pp. 139-146, 2010.

T. Horikawa, An approach for scalability-bottleneck solution: identification and elimination of scalability bottlenecks in a DBMS, ICPE '11 Proceeding of the second joint WOSP/SIPEW international conference on Performance engineering, pp. 31-42, 2011.

T. Horikawa, An Unexpected scalability bottleneck in a DBMS: A hidden pitfall in implementing mutual exclusion, to be published in the 23rd IASTED International Conference on Parallel and Distributed Computing and Systems, 2011.

cf. 最初の3本はMySQLを対象としています(orz). 4本目はPostgreSQLを扱っています.

Empowered by Innovation

**NEC**