

# Inner look on Oracle latches

**A Technical White Paper**

**By**

**David Gornshtein**

**Boris Tamarkin**

**[WisdomForce Technologies, Inc](http://www.wisdomforce.com)**

**<http://www.wisdomforce.com>**

**Revision 1.1**

**Note\*:** The latest copy of the document is available at  
[http://www.wisdomforce.com/dweb/resources/docs/internals\\_latches.pdf](http://www.wisdomforce.com/dweb/resources/docs/internals_latches.pdf)

*This topic is in-depth explanation of what is Oracle latch, how it is working, what is its implementation, how it presented in Oracle SGA and so on.*

Disclaimer .....	2
Preface.....	2
What is Oracle latch.....	2
Latches and enqueues .....	3
Oracle latch internals .....	3
How process sleep/wakeup mechanism is working for the process waiting on latch .....	8
Latch contention identification .....	8
Problems with latches .....	12
1. Library cache related.....	12
2. Buffer cache related latches .....	14

## **Disclaimer**

All presented tests shall NOT be done on production database without proper testing.

## **Preface**

*The information provided in this paper is kind of internals info in nature and presumed to be likely invisible to the user community. However it is always useful for professional to know how Oracle latch is working in case of any issues arise.*

We will try avoiding duplicating of information from metalink note 22908.1 which is publicly available in internet on several sites. We will try also to avoid duplication of information from Steve Adams topic about the latches from his "Oracle8i Internal services" book. By the way this book is highly recommended to read, since it is not only informative but also very useful, especially combined with [www.ixora.com.au](http://www.ixora.com.au) APT scripts. However some overlapping might will take place, since both metalink note and Steve Adams are very comprehensive and will help presenting complete view.

## **What is Oracle latch**

Accordingly to Oracle documentation, latch is a type of lock that can be very quickly acquired and freed. Latch typically used for preventing more than one process from executing the same piece of code at a given time. However, Oracle latch is actually a spin mutexes on process level which used

in most cases to guard the SGA fixed and heap based data structures from being changed every time by more than just a single process.

## Latches and enqueues

Enqueues are used usually to prevent more than one process from accessing the same data structure at a given time. Latch present a contrast with enqueues which always protect structures which has some disk based equivalent such as database objects, data blocks or table rows.

Another significant difference between latch and enqueue is that enqueue has ordered queue of waiters serviced in FIFO while processes compete for latch serviced in the arbitrary order, e.g. you can see list of enqueue waiters and know the exact order of service for those waiters.

On the other hand, latch has a pointer to processes that went to sleep while waiting actively for that latch. “Actively” meaning spin (willing-to-wait) or trying to get into immediate (no-wait) mode. The mechanism of holding process which gone to sleep while waiting for latch is called “latch wait posting” and controlled by init.ora parameter “*\_latch\_wait\_posting*” which is true by default. Number of times that willing-to-wait will spin on mutex before **yield** CPU and go to sleep predefined by init.ora *\_spin\_count* parameter to wakeup these processes after latch has been freed by the process that currently uses it.

## Oracle latch internals

Server latch implementation called KSL, e.g. every table with name starting with x\$ksl... is latch related and every SGA has ksl... structure as well. In order to get complete list of x\$ksl table, connect with sysdba permissions to oracle and run query:

```
SELECT kqftanam
FROM x$kqfta
WHERE kqftanam LIKE 'X$KSL%'
```

There are two structures associated with a latch in KSL - kslt and kslla.

struct kslt is a definition of the latch itself and all these structures (latches itself and pointers to hold latches v processes) are visible on instance level via fixed SGA, while kslla is a process related structure and is visible as part of X\$KSUPR structure.

All available kslla structures can be identified by using the following query:

```
SELECT c.kqfconam field_name, c.kqfcooff offset, kqfcotyp,
       DECODE (kqfcotyp,
              0, 'hex string',
              1, 'varchar2',
              2, 'number',
              11, 'word',
              12, 'datetime',
              23, 'raw',
              'other'
              ) typ,
       kqfcosiz sz
FROM x$kqfco c, x$kqfta t
```

```

WHERE t.indx = c.kqfcotab
      AND kqftanam = 'X$KSUPR'
      AND kqfconam LIKE 'KSLLA%'
      AND c.kqfcooff > 0
ORDER BY offset ASC

```

For example, if you are using Oracle 9.2.0.5 you will see the following results

```

KSLLLALAQ,296,0,hex string,8
KSLLLAWAT,320,0,hex string,8
KSLLLAWHY,328,11,word,8
KSLLLAWER,336,11,word,4
KSLLLALOW,344,0,hex string,4
KSLLLASPN,384,0,hex string,8
KSLLLAPRV,544,11,word,4
KSLLLAPSN,548,11,word,4
KSLLLAPSC,552,11,word,4
KSLLLAPRC,556,11,word,4

```

Only kslawat and kslaspn are actually available via V\$PROCESS; kslawat represents latch waits and kslaspn represents latch spins. Other fields can be used mostly for debugging purposes such as halt analysis, etc. For example, kslalAQ has pointer kslt\* to acquired latch set by KSLBEGIN, pointing to the head of linked list used in latch wait posting implementation (KSLBEGIN is a begin macro in the defined in the Oracle ksl.h code). Once a process acquires a latch at a certain level, it cannot acquire anymore latch at level that is equal or less than that level (unless it acquires nowait).

To support this issue, another non documented column kslallow from X\$KSUPR is used. To see all the latches in the fixed SGA, the following query can be used:

```

select k.ksmfsadr, ksmfsnam, ksmfstyp, ksmfssiz, ksllldnam, ksllldlvl
from x$ksmfsv k, x$kslld a
where k.ksmfstyp like '%ksllt%' and k.ksmfsadr = a.kslldadr
order by ksmfsnam

```

Let's see how we can lock / unlock enqueues latch ksqueql\_ via oradebug.

This latch is used to protect operation on KSE (enqueue) related structures in memory. Assume, value of k.ksmfsadr of this latch that received from the previous query was 000000038000A0C0 (0x000000038000A0C0 = 15032426688)

In order to lock the latch, we can use function

```

word kslgetl(ksllt *latch_to_lock, word wait)
sqlplus "/ as sysdba"
oradebug setmypid
SQL> oradebug call kslgetl 15032426688 1

```

Function returned the value 1, meaning that we locked the latch.

Now let's try to connect to Oracle. You can see that your session was halt because you are holding enqueue latch, therefore Oracle is unable to update even its own X\$ table entries.

Let's return to oradebug and will release (free) the enqueue latch.

```

SQL> oradebug call kslfre 15032426688Function returned 8000A0C0

```

This time yours another session continued immediately.  
Now let's check the enqueue latch by querying v\$latch:

```
select wait_time
from v$latch
where name = 'enqueues'
```

The wait time returned is extremely big (in example it was 619714031, e.g. 10 minutes).  
List of all latch related tables as following:

```
GV$LATCH
GV$LATCH_CHILDREN
GV$LATCH_PARENT
GV$DLM_LATCH
```

**\*Note:** list excludes x\$kslld which is process internal structure and not represented in the SGA at whole.

GV\$DLM\_LATCH is a special fixed table, used in the OPS/RAC environment only to monitor distributed lock manager latch. This fixed view exists only for historical reasons, since DLM did not use KSL ksltt and kslla latch related structures prior to version 8.1.5. DLM had its own ksltt and kslla structures. DLM uses standard KSL structures from version 8.1.5 and up, therefore DLM latches can be monitored via V\$LATCH fixed views.

GV\$LATCHHOLDER is a process, e.g. X\$KSUPR, fixed array of process structures,  
GV\$LATCH\_MISSES is table which is non-directly points to the latch structures.

The next interesting question is where in the Oracle code each latch is used. It is possible to see the latch contention on some latch. However its name would be meaningless so at least you can identify where in Oracle code (up to function name) either this latch has been locked.

Such identification can be done, in normal case by running the following query:

```
select parent_name, location, nwfail_count, sleep_count from v$latch_misses;
```

where column location divided to 1 - 3 parts divided by ':'

- 1) Oracle kernel function.
- 2) Optional kslbegin (macro to lock latch) or operation name.
- 3) Optional description or comment if single function has several locks/unlocks for the same latch.

For example, let's look at the function "kcbgtr" used to "get cache buffer for consistent read" or in other words it reads buffer from disk to buffer cache and then rolls it back up to first SCN of the query, caused CR read.

In Oracle 9.2.0.5 this function has 4 different places where "cache buffers lru chain"<sup>1</sup> latch to which the appropriate related block can be locked.

Execute the following query:

```
SELECT t1.ksllasnam "parent_name", t2.ksllwnam "location",
       t2.ksllwbl "unit to guard"
FROM x$ksllw t2, x$kslws t1
WHERE t2.indx = t1.indx
      AND t2.ksllwnam LIKE 'kcbgtcr%'
      AND ksllasnam = 'cache buffers lru chain'
```

Result of query above will be something like:

parent_name	location	unit to guard
cache buffers lru chain	kcbgtcr:CR Scan:KCBRSTOP	buffer header
cache buffers lru chain	kcbgtcr:CR Scan:KCBRSERVE	buffer header
cache buffers lru chain	kcbgtcr:CR Scan:KCBRSKIP	buffer header
cache buffers lru chain	kcbgtcr:CR Scan:best	buffer header

*\*Note: In next topic about buffer cache and buffer cache related latches in the future discussion we will talk about CR blocks and related parameters.*

If Oracle instance is halt and you have some reason to think that is caused by latching problem, then you can use oradebug to dump latch statistics:

1. `connect as sysdba`
2. In order to dump latch states, perform the following operations:
  - `oradebug setmypid`
  - `oradebug dump latches 2`
  - `oradebug tracefile_name`

Trace file will be generated with name, for example such as  
`/oravl01/oracle/adm/bigsun/udump/bigsun_ora_21039.trc`

Open this trace file to see the latch with high and constantly increasing between dumps sleeps count in the case of willing-to-wait latch and failed count in the case of no-wait latch.

For example, if you performed 2 dumps with insignificant interval between and have seen in the first dump for some child redo allocation latch sleeps count of 3. In the second dump sleeps count

---

<sup>1</sup> specific cache buffers lru chain latch is guard for specific part of buffer lru chain and block we are going to use has entry in this hash chain

of 13 with "failed first" increased to the same number as sleeps count and "gotten" counts remains the same. All this means that some (and at least one) process is waited constantly for this latch all the time between two dumps, e.g. another process hold this latch.

Example:  
dump (1):

```
396670178 Child redo allocation level=5 child#=1
  Location from where latch is held: kcrfwi: more space: strand #
  Context saved from call: 0
  state=free
  gotten 7776125 times wait, failed first 355 sleeps 3
  gotten 0 times nowait, failed: 0
```

dump (2):

```
396670178 Child redo allocation level=5 child#=1
  Location from where latch is held: kcrfwi: more space: strand #
  Context saved from call: 0
  state=free
  gotten 7776125 times wait, failed first 365 sleeps 13
  gotten 0 times nowait, failed: 0
```

Let's try and emulate such "bug" example. To do that, we will need two sessions namely sess\_1 and sess\_2. Our child redo allocation latch address is 0x396670178 that can be converted into 15408234872. sess\_1 will be connected as sysdba and sess\_2 as oracle dba:

```
step 1. sess_1: oradebug setmypid
step 2. sess_1: oradebug dump latches 2
step 3. sess_1: oradebug call kslgetl 15408234872 1
step 4. sess_2: create table test1 as select * from dba_objects;
step 5. wait 1 minute
step 6. sess_1: oradebug call kslfre 15408234872
step 7. sess_1: oradebug dump latches 2
```

On the step 4 session sess\_2 was halted. If you will try performing dump latches while some latch is locked, you will have a good chance to receive well known ORA-03113 (end-of-file on communication channel). However this is relates to latch and platform dependent.

Due to the error stacks it is looks like this is Oracle bug since latch state represented in the latch dump. However during all tests we have performed, we never seen in the dump files states other than "state=free". It's seems that Oracle waits several seconds until each latch will be freed and then dumps its state. If latch has not been freed during several seconds, ORA-03113 may occur...

```
396670178 Child redo allocation level=5 child#=1
  Location from where latch is held: kcrfwi: more space: strand #
  Context saved from call: 0
  state=free
  gotten 7784997 times wait, failed first 362 sleeps 10
  gotten 0 times nowait, failed: 0
```

```

396670178 Child redo allocation level=5 child#=1
      Location from where latch is held: kcrfwi: before write: strand #
      Context saved from call: 0
      state=free
gotten 7786065 times wait, failed first 367 sleeps 14
gotten 0 times nowait, failed: 0

```

In this case "gotten" increased significantly since to perform following statement:

```
create table test1 as select * from dba_objects;
```

more then 1000 times redo the space allocations has been performed.

In the general case, even if your instance is not halt but you want to see latch statistics in this format, you can dump latches statistics by running

```
ALTER SESSION SET EVENTS 'immediate trace name latches level 2';
```

Dump latches have two available levels:

- level 1 when dump just basic latch information (without statistics)
- level 2 when dump just latch information with statistics

## How process sleep/wakeup mechanism is working for the process waiting on latch

If process will not wait for the latch due to single processor machine, or after spinning up to `_spin_count`, process going to sleep. Prior to moving to sleep stage, the process will start signaling scheduler that shall wakeup process by using exponential backoff per miss histogram.

From `x$ksllt` fixed view is possible to see that Oracle latches using array of four buckets, from `ksllthst1` to `ksllthst4`, where each bucket is unsigned integer.

The sums of sleeps can be monitored by `V$LATCH` `sleep1` to `sleep4` columns.

It is possible to calculate next sleep time by looking at the `x$ksllt.ksllthst1` to `x$ksllt.ksllthst4`:

```
ksllthstX sleep = 2^((bucket+1)/2) - 1 + (ksllthst(X - 1) + ... + ksllthst1)
```

e.g. actual sleep time corresponding to each bucket in the histogram is about

```
2** (bucket+1)/2 - 1 plus sleep time of all lower buckets.
```

Buckets from 5 to 11 have been used in early versions of Oracle 7 and not used anymore.

All of max exponential backoffs greater than the highest bucket are added to the highest bucket.

However, the parameter `_max_exponential_sleep` in `init.ora` defines maximum sleep time that can be reached using exponential backoffs mechanism.

## Latch contention identification

So, what are choices besides halting my Oracle instance by locking enqueue (or any other) latch? First thing to try is validate which part of your wait events caused by latch contention. How to do that? Discard all event that you can not avoid by server tuning, e.g. SQL\*Net events indicates client/server network round trips and waits for client, pmon and smon related events, various "timer" related events and also "Null event". "Null event" is a special case of wait that sometimes may indicate real wait; however it is not easy to indicate what it waiting for. In Oracle starting 9.2.0.1 up to 9.2.0.3 there was significant number of Null events that was not commented or defined. Since 9.2.0.4 almost all Null events has been changed to one of the documented events type.

In addition, rdbms ipc message and gcs remote message are non related waits since these wait events indicate IPC communication between oracle processes wait events in local instance and RAC respectively.

In general case, the following query will return more then 99.5% of all wait events

```

SELECT event, total_waits, time_waited, percent
  FROM (SELECT se.event, se.total_waits, se.time_waited,
              (se.time_waited / total.total_waiptime) * 100 percent
        FROM (SELECT sum (time_waited) total_waiptime
              FROM v$system_event
              WHERE event NOT IN
                   ('pmon timer',
                   'ges pmon to exit',
                   'ges lmd and pmon to attach',
                   'ges cached resource cleanup',
                   'parallel recovery coordinator waits for cleanup of
slaves',
                   'smon timer',
                   'index (re)build online cleanup',
                   'dispatcher timer',
                   'dispatcher listen timer',
                   'timer in sksawat',
                   'SQL*Net message to client',
                   'SQL*Net message to dblink',
                   'SQL*Net more data to client',
                   'SQL*Net more data to dblink',
                   'SQL*Net message from client',
                   'SQL*Net more data from client',
                   'SQL*Net message from dblink',
                   'SQL*Net more data from dblink',
                   'SQL*Net break/reset to client',
                   'SQL*Net break/reset to dblink',
                   'PL/SQL lock timer',
                   'rdbms ipc message',
                   'ges remote message',
                   'gcs remote message',
                   'jobq slave wait',
                   'Null event'
                )
              AND time_waited > 10) total, v$system_event se
        WHERE se.event NOT IN
              ('pmon timer',
              'ges pmon to exit',
              'ges lmd and pmon to attach',
              'ges cached resource cleanup',
              'parallel recovery coordinator waits for cleanup of
slaves',

```

```

        'smon timer',
        'index (re)build online cleanup',
        'dispatcher timer',
        'dispatcher listen timer',
        'timer in sksawat',
        'SQL*Net message to client',
        'SQL*Net message to dblink',
        'SQL*Net more data to client',
        'SQL*Net more data to dblink',
        'SQL*Net message from client',
        'SQL*Net more data from client',
        'SQL*Net message from dblink',
        'SQL*Net more data from dblink',
        'SQL*Net break/reset to client',
        'SQL*Net break/reset to dblink',
        'PL/SQL lock timer',
        'rdbms ipc message',
        'ges remote message',
        'gcs remote message',
        'jobq slave wait',
        'Null event'
    )
    AND se.time_waited > 10
    ORDER BY percent DESC)
WHERE ROWNUM < 10

```

When you see now that latch related wait events presenting significant part of your waits, you need to perform drill down to investigate your latching problem.

You can start by running standard query of misses\_ratio and immediate\_misses\_ratio:

```

SELECT inst_id, name, immediate_gets, immediate_misses,
       decode (immediate_gets + immediate_misses, 0, 0,
              immediate_misses / (immediate_gets + immediate_misses) * 100)
immediate_misses_ratio
FROM   gv$latch
where  immediate_misses > 0
order by immediate_misses_ratio desc;

```

```

SELECT inst_id, name, gets, misses, immediate_gets, immediate_misses,
       decode (gets, 0, 0, misses / gets * 100) misses_ratio
FROM   gv$latch
where  misses > 0
order by misses_ratio desc;

```

Lets look at latches itself, which latch contention is important and which not. Most of the latches are singles and can be tuned just by changing system wide init.ora `_spin_count` parameter. Changing `_spin_count` parameter in most cases will have very insignificant effect on whole system. The only case have seen by author that setting `_spin_count` parameter really improved performance, was the performance benchmark on 2 64 CPU HP Superdome servers and Oracle RAC, where we had very significant latch contention on "library cache", "cache buffers chains" and "redo allocation" latches.

Anyway, even in this case using HP-UX NOAGE scheduler will improve performance in better way than just increasing the value of `_spin_count` system wide. However, if you have HP Superdome and have latch contention, then you may try increase `_spin_count` up to 5000, or even

to 20000. This can help when setting HPUX\_SCHED\_NOAGE in the init.ora. As example, look at the Oracle9i Administrator's Reference, topic Tuning for Oracle9i on HP ([http://download-west.oracle.com/docs/html/A97297\\_01/appb\\_hp.htm](http://download-west.oracle.com/docs/html/A97297_01/appb_hp.htm)). In addition to looking at system wide gv\$latch statistics, you can also try identifying latch contention by using tracing sessions by event 10046, level 4.

## Problems with latches

Latches and latching problem could be described only in context of the structures of the latches guards. Therefore we will give a little bit background about the shared pool along with library cache latching problems, with cache chain latches, about the buffer cache, etc.

### 1. Library cache related.

If you have extensive SQL parsing, you will see a lot of 'library cache' or kgl related events. In addition, even if your application almost does not perform parsing, but does perform a lot of connections running the same queries, you can see some contention on "library cache lock". But you won't see contention on the "library cache pin" which is used on statement parsing. These statistics may be improved a little by changing init.ora CURSOR\_SHARING parameter from EXACT to FORCE or SIMILAR and CURSOR\_SPACE\_FOR\_TIME parameter to be set to TRUE.

However, much preferable to reuse cursors in application - use OCI, OTL, OCCI or even PROC with close\_on\_commit=no preprocessing parameter. You may think to increase number of kgl latches by setting value \_kgl\_latch\_count in init.ora. Unfortunately as from our experience it will not help much. \_kgl\_latch\_count should be the prime number, and you can check current number of these latches by running:

```
SELECT count(*)
FROM v$latch_children
WHERE NAME = 'library cache';
```

Then you can change it by the following command, for example, to 11 (or another prime number):

```
alter system set "_kgl_latch_count" = 11 scope=spfile sid='*'
```

And afterwards restart all related instances. The default for \_kgl\_latch\_count is 0, and default number of kgl latches is next prime number greater than or equal to number of CPUs.

We would like also to describe the \_kgl\_bucket\_count setting which is not directly related to shared pool latches, but it is a part of DBA's intent to set it to the higher value, mostly without any reason. This parameter defines the number of object buckets in the shared pool. Every bucket can grow and become quite large. These buckets created with initial size of 509 objects and can grow up to  $509 * 2^8 - 1$  each. This means, that initially bucket has 509 slots and can be extended twice in size up to 7 times. E.g. if you will specify \_kgl\_bucket\_count = 2, you will have actually up to  $2^8 * 8 * 2 - 1 = 4095$  standard shared pool object buckets and each bucket can hold 509 objects.

Number of KGL objects you have pinned in your shared pool can be found by:

```
select count(*)
from x$kglob
where kglhdadr = kglhdpar
```

There is a good chance that you will have no more than several thousands of objects which can be easily hold event with \_kgl\_bucket\_count = 1. In Oracle 9.2.0.x, for example, default value for

`_kgl_bucket_count` is 9. This means, that you should never touch this parameter, until you will be able to explain why do you need change this setting.

Steve Adams provides script calculating the number of KGL buckets sufficient enough to avoid dynamic extension of each KGL buffer in runtime. However, such extension is very fast operation and your chance to catch some wait on this by using Oracle system wide statistics is close to zero.

It is possible to see all KGL latches and buckets by dumping library cache via `alter session` or `oradebug`. If you want to see just latches and basic shared pool statistics, then use level 3. Otherwise use level 10 in case you want to see KGL object handles in buckets.

```
alter session set events 'immediate trace name library_cache level 10';
```

`library_cache` dump on level 3 is useful and provides LIBRARY CACHE STATISTICS by namespace in the readable form, something like

namespace	gets	hit ratio	pins	hit ratio	reloads	invalids
CRSR	9671	0.964	38372	0.985	23	0
TABL/PRCD/TYPE	1856	0.842	9576	0.941	0	0
BODY/TYBD	10395	0.999	10394	0.999	0	0
TRGR	3	0.333	3	0.333	0	0
INDX	170	0.741	95	0.537	0	0
CLST	281	0.972	373	0.973	0	0
OBJE	0	0.000	0	0.000	0	0
PIPE	0	0.000	0	0.000	0	0
LOB	0	0.000	0	0.000	0	0
DIR	0	0.000	0	0.000	0	0
...						

```
alter session set events 'immediate trace name library_cache level 3';
```

Level 3 provides with much more information than `v$sqlibrarycache`. Output is easier to understand than `x$kgllst`.

Similar information but might less in depth to that you can see by running following query:

```
select *
from v$sqlibrarycache
```

`library_cache` dump on level 3 contains the same kind of information with exception this information is better readable than can be received by query:

```
select *
from x$kgllst
```

Dump on level 10 is almost useless unless you have some ORA-600 that looks like bug and would like to identify what the problem is.

## **2. Buffer cache related latches**

To be continued ....