New features of Latches and Mutexes in Oracle 12c.

Andrey Nikolaev
Oracle Database Performance Expert

RDTEX
Russia

09 December 2015

Who am I

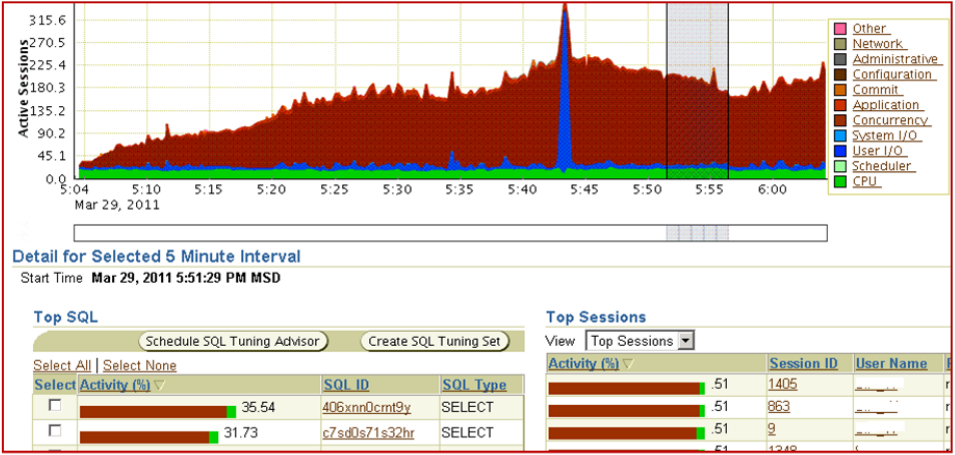
- Andrey.Nikolaev@rdtex.ru
- RDTEX, First Line Support Centre.
- http://andreynikolaev.wordpress.com
  "Latch, Mutex and Beyond"
- Specialist in Oracle performance tuning.
- Over 25 years of Oracle-related experience as a research scientist, developer, DBA, performance consultant, and lecturer ...
- In late 80s took part in the first applications of Oracle databases in Particle Physics.
- Occasionally present at conferences.
- A member of Russian OUG:
- Background in physics and mathematics and member of the American Mathematical Society (AMS).

The Goal of this Presentation

- Latch and mutex contention is still one of the most complex DBA challenges.
- I discussed the internals of Oracle 8i-11g latches and mutexes at UKOUG2011 and Hotsos2012. My research have found that:
  - Exclusive latches spin for 20,000 cycles.
  - The exponential backoff disappeared in Oracle 9.2.
  - The **latch free** wait may be infinite.
  - Mutexes have a variety of spin-and-wait schemes.
- Now, I will compare the Oracle 12c latches and mutexes to show that:
  - The mutexes are not a replacement for the latches. They operate at different timescales and have different purposes.
  - There are ways to monitor and tune Oracle latches and mutexes.
  - Sometimes it is worthwhile to adjust the **_spin_count**.

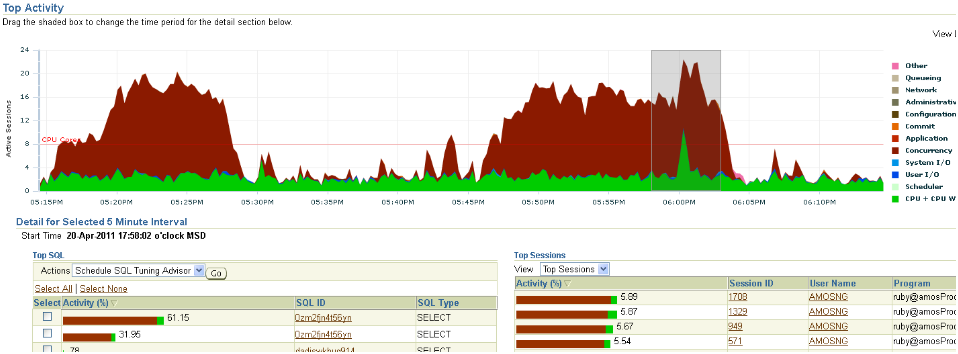
Episode of a Latch Contention

- Oracle instance hanging caused by heavy **cache buffers chains** latch contention:



Episode of a Mutex Contention

- Incidents of **cursor: mutex S** contention caused by high version counts of frequently executed SQL operators:



Disclaimer

Production use of the undocumented techniques described here should always be approved by Oracle Support.

Oracle technologies evolve rapidly. This presentation discusses the latches and mutexes as of Oracle 12.1.0.2.

Additional Info about Oracle Latches and Mutexes

- "**Oracle8i Internal Services for Waits, Latches, Locks, and Memory**" by Steve Adams, 1999.
  - Founded new era of Oracle performance tuning.
- "**Systematic Latch Contention Troubleshooting**" approach by Tanel Poder, 2010.
  - **Latchprofx.sql** script revolutionised the latch tuning.
- "**Oracle Core: Essential Internals for DBAs and Developers**" by Jonathan Lewis, 2011.
- My blog "**Latch, Mutex and Beyond**" `http://andreynikolaev.wordpress.com`.

Serialization mechanisms in Oracle

Oracle Database Concepts 12c:

- "A latch is a simple, low-level serialization mechanism that coordinates multiuser access to shared data structures, objects, and files."
- "A mutual exclusion object (mutex) is a low-level mechanism that prevents an object in memory from aging out or from being corrupted ..."
- "Internal locks are higher-level, more complex mechanisms ..."

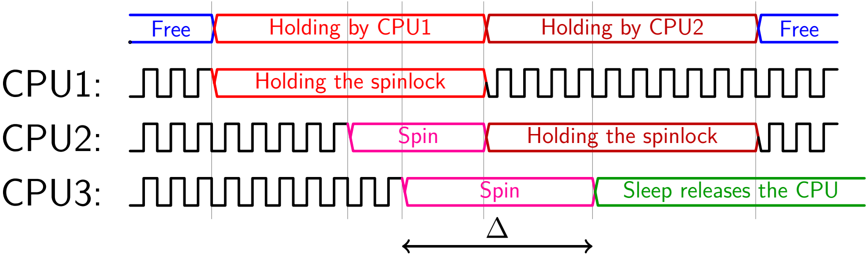
	Locks:	Latches:	Mutexes:
Access	Several Modes	Types and Modes	*Operations*
Acquisition	FIFO	SIRO (spin) + FIFO	*SIRO (spin)*
Atomic	No	Yes	*Yes*
Timescale	Milliseconds	Microseconds	*SubMicroseconds*
Lifecycle	Dynamic	Static	*Dynamic*

Classic Spinlocks

- Oracle latches and mutexes are spinlocks.
- According to Wikipedia: "Spinlock waits in a loop repeatedly checking until the lock becomes available ...". Spinlocks use atomic instructions.
- Spinlocks were first introduced by Edsger Dijkstra in 1965, and there use has since been thoroughly researched.
- Many sophisticated spinlock realizations have been proposed and evaluated including the TS, TTS, MCS, Anderson, and more.
- There are two general types of spinlocks:
  - System spinlocks. Kernel OS threads cannot sleep and must spin until success.
    Metrics: Atomic operations frequency and shared bus utilisation.
  - User spinlocks. Average holding time of Oracle latches and mutexes is about 1 us. It is more efficient to poll a lock rather than preempt the thread doing 1 ms context switch.
    Metrics: CPU and elapsed times.

How the Spinlock Works

The spinlock location:



- Oracle latches and mutexes:
  - Use atomic hardware instruction for the immediate get.
  - If missed, the process repeatedly polls the spinlock location during spin.
  - The number of spin cycles is limited by spin count.
  - If spin get does not succeed, the process sleeps.
- Oracle counts the gets and sleeps and we can measure Utilisation.

Spinlock Realizations

	Pseudocode:	Problems:
TS. pre-11.2 mutex	**while(Test_and_Set(lock));**	Shared bus saturation
TTS. Oracle latch, 11.2 mutex	**while(lock‖Test_and_Set(lock));**	Invalidation storms on release ("open door").
Anderson, MCS, etc.	Queues. Widely used in Java, Linux kernel ...	CPU and memory overhead, preemption issues

Tools

Oradebug: Internal Oracle Debugger

- Oradebug call. Allows us to invoke any internal Oracle function manually.

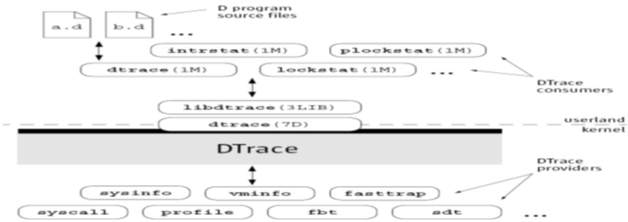
```
SQL> oradebug call kslgetl 0x5001A020 1 2 3
Function returned 1
```

- Oradebug peek. Examines contents of any memory location.

```
SQL> oradebug peek 200222A0 24
[200222A0, 200222B8) = 00000016 00000001 000001D0 00000007
```

- *Oradebug poke. Modified memory. No longer works in 12c.*
- Oradebug watch. Sets a watchpoint on a region of memory.
- Oradebug event wait_event["**latch free**"] trace("%s\n",shortstack())

DTrace: Solaris Dynamic Tracing Framework



DTrace made possible to investigate how the spinlocks perform. It allowed me:

- Create triggers on any event or function call in Oracle and Solaris.
    - **provider:module:function:name**
    - **pid1910:oracle:kslgetl:entry**
    - **pid1910:oracle:kgxExclusive:entry**
- Write trigger bodies - actions. DTrace can read and change any memory location.
- Count the spinlock spins, trace waits, perform experiments.
- Measure times and distributions up to microsecond precision.

Latch Contention Testcases

- My investigation is based on testcases.
- A contention for a shared **cache buffer chains** latch occurs when sessions scan concurrently a block with multiple versions:

```
create table latch_contention as select rownum id from dba_objects
where rownum<100;  ...
update latch_contention set id=id+100 where id=<thread number>;
@sample 1 latch_contention 1=1 1000000 ...
```

- Frequent concurrent hash joins cause a contention for exclusive **row cache objects** latches (bug 13902396):

```
for i in 1..1000000 loop
  select /*+ use_hash(a) */ 1 into j from dual a natural join dual;
end loop;
```

- Many other contention scenarios are possible.

Mutex Contention Testcases

- **Cursor: pin S** mutex contention arises when the same SQL operator is executed concurrently at high frequency.

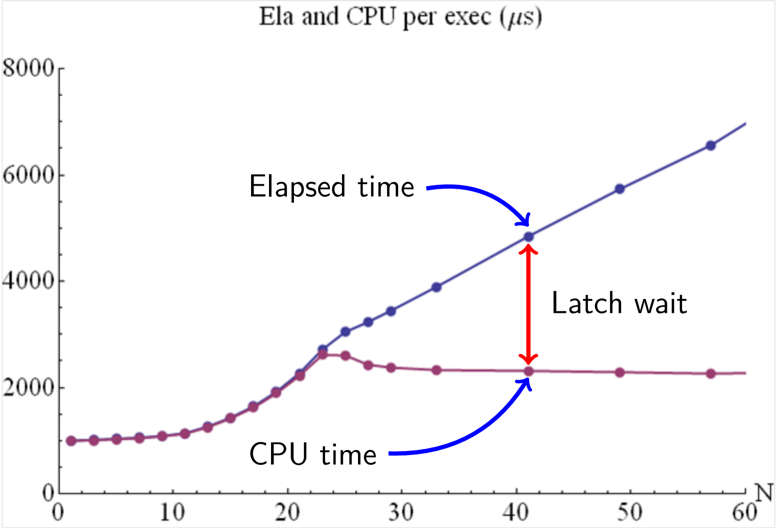
```
for i in 1..1000000 loop
  execute immediate 'select 1 from dual where 1=2';
end loop;
```

- **Cursor: mutex S** waits appear after the addition of several versions of the SQL and setting **session_cached_cursors**=0.
- **Library cache: mutex X** contention arises when anonymous PL/SQL block is executed concurrently at high frequency.

```
for i in 1..1000000 loop
  execute immediate 'begin demo_proc();end;';
end loop;
```

- Many other mutex contention scenarios are possible.

Row Cache Latch Contention Testcase



Elapsed time and CPU time vs. number of threads.
Oracle Database Appliance (24 SMT).

How Oracle processes and sessions hold latches and mutexes

Latches in State Objects Dumps

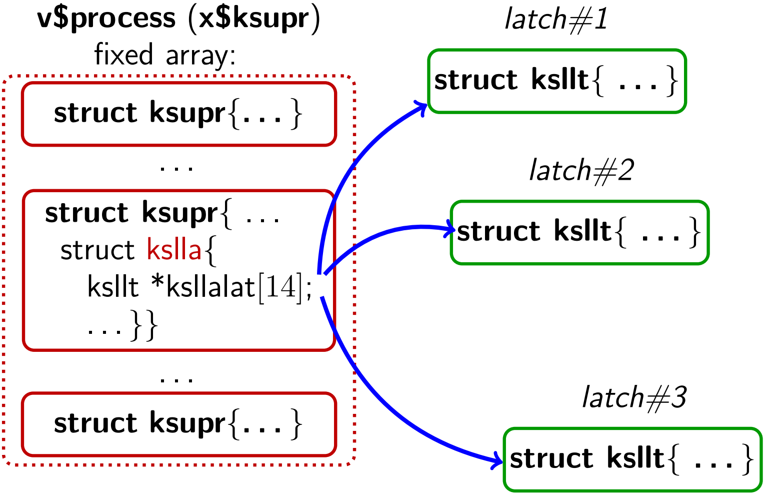
- **Note 423153.1**. *Reading and Understanding Systemstate Dumps*. Example: A process with the PID=46 is waiting for a latch:

```
PROCESS 46:
... (latch info) hold_bits=0x0
      Waiting For:
        0x6000aeb0 'test excl. non-parent lmax' (level=8)
     ... Waiter Location: ksfq2.h LINE:607 ID:ksfqpaa:
         Waiter Context: 100...
...   Current Wait Stack:
     0: waiting for 'latch free'
        address=0x6000aeb0, number=0x8, tries=0x0 ...
```

- Another Oracle process with the PID=7 is holding the latch:

```
... (latch info) hold_bits=0x100
      Holding:
        0x6000aeb0 'test excl. non-parent lmax' (level=8)
        Holder Location: ksfq2.h LINE:607 ID:ksfqpaa:
        Holder Context: 100 ...
```

Latches Held by the Processes



- Each process has an array of references to the latches it is holding.
- This **kslla** structure is embedded into the process state object.

KGX Mutexes in State Object dumps

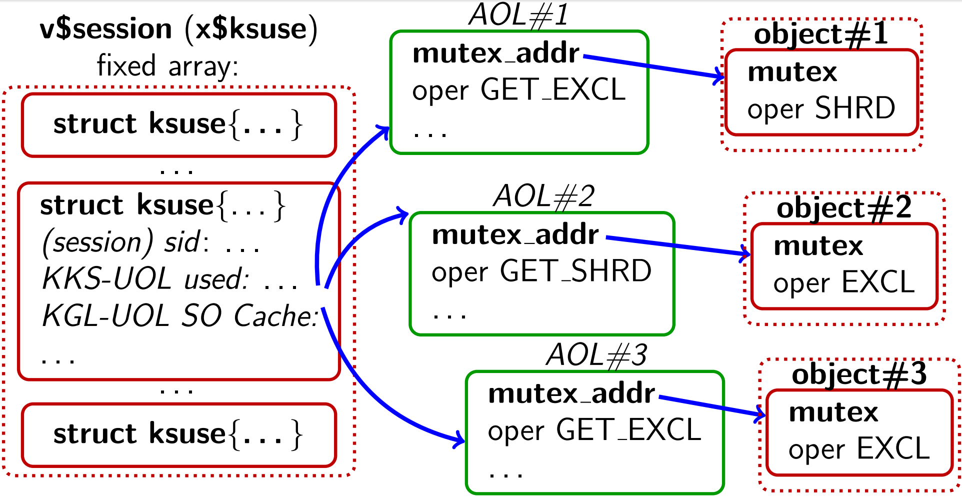
- **Note 423153.1**. *Reading and Understanding Systemstate Dumps*.
- Example: An Oracle session with the SID=22 is holding a **Cursor: pin** mutex in **E** mode during a SHRD_EXAM operation:

```
KGX Atomic Operation Log 3ea866010
      Mutex 3f119b5a8(22, 1)  idn 382da701 oper SHRD_EXAM
      Cursor Pin uid 22 efd 0 whr 5 slp 0 ...
```

- The mutex **identifier** (**idn**) is the hash value of the current SQL.
- Below, another session with SID=24 is waiting for the mutex during the GET_SHRD operation:

```
... waiting for 'cursor: pin S'
    idn=0x382da701, value=0x1600000001, where=0x500000000
... KGX Atomic Operation Log 3ea8d8c08
       Mutex 3f119b5a8(22, 1) idn 382da701 oper GET_SHRD
       Cursor Pin uid 24 efd 0 whr 5 slp 46685 ...
```

Mutexes Held by the Sessions



- Each session has an array of references to the Atomic Operation Logs (AOL's) that it is using.
- The mutexes themselves are embedded into the KGL objects.

Atomic Operation Log (AOL) Structure

- The session changes the mutex state using an AOL:

```
KGX Atomic Operation Log 3ea866010
       Mutex 3f119b5a8(22, 1) idn 382da701 oper SHRD_EXAM
       Cursor Pin uid 22 efd 0 whr 5 slp 0 ...
```

- The AOL contains information about a mutex operation in progress.
- In order to change the mutex state, the session should:
  - Allocate the AOL structure.
  - Fill the AOL with the data about the mutex and the desired operation.
  - Execute a mutex acquisition routine.

```
SQL> oradebug peek 0x3EA866010 12
[3EA866010, 3EA86601C) = 3F119B5A8 00050703 00000016 ...
                            Mutex   whr   op   uid(sid)
```

- AOL's in systemstate dump show all mutex operations in progress.
- In case of a session failure AOL's are used by the PMON during a recovery.

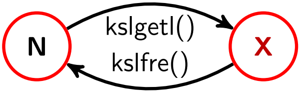
RDTEX TECHNOLOGY CONSULTING

State diagrams, instrumentations, and interface routines

DTrace Reveals Latch Interface Routines

- Oracle calls the following functions to acquire the exclusive latch:

kslgetl(laddr, wait, why, where) – get the exclusive latch.
kslg2c(l1,l2,trc,why, where) – get two exclusive child latches.
kslgpl(laddr,comment,why,where) – get the parent and all children.
kslfre(laddr) – free the latch.

- *Oracle allows us to do the same using the* ***oradebug call***.
- Exclusive latch state diagram: 

- In Oracle 12c, the first word of the busy latch contains the PID of the holder process:

```
SQL> oradebug peek 200222A0 24
[200222A0, 200222B8) = 00000016 00000001 000001D0 00000007
                       pid^     gets     latch#   level#
```

Where and Why Oracle Gets the Latch

To request the latch, the Oracle kernel routine needs:

- **laddres** – address of the latch in the SGA.
- **wait** – flag for no-wait (0) or wait (1) latch acquisition.
- **where** - code for the location from where the latch is being acquired. Oracle enlists possible **where** values for the latch in the **x$ksllw** fixed table.
- **why** - context to explain why the latch is being acquired at this **where**. It contains the DBA address for the **CBC** latches (Tanel Poder), the SGA chunk address for the **shared pool latch**, session address, etc.
- The latch gets are instrumented by the **where** values in the **v$latch**, **v$latchholder**, **v$process**, and **v$latch_misses** fixed views.
- Tanel Poder introduced the high-frequency sampling of the **why** values from **v$latchholder** to systematically troubleshoot the latch contention.

Shared Latches

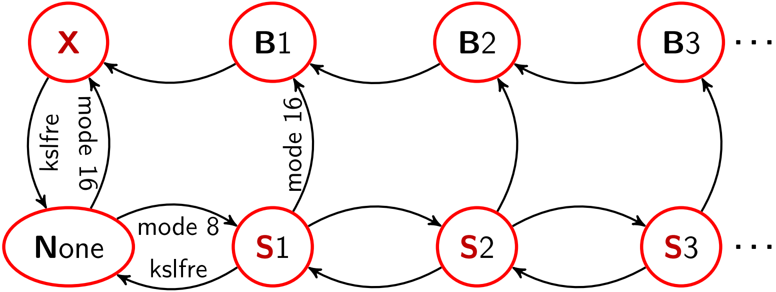
- Shared latches are Oracle's realization of the "Read-Write" spinlocks.
- The **S** and **X** modes of the shared latch are incompatible.
- **kslgetsl_w(laddr,wait,why,where,mode)** – get the shared latch.
  8 – Shared mode, 16 – eXclusive mode
- *Internally, however, Oracle uses the **ksl_get_shared_latch()** function with six arguments.*
- In **S** mode, the latch memory location represents the number of processes that are holding the latch simultaneously. For example:

```
SQL> oradebug peek 0x6000AEA8 24
[6000AEA8, 6000AEC0) = 00000002 00000000 00000001 00000007
                        ^Nproc   ^X flag   gets     latch#
```

- The **cache buffers chains** shared latches are also acquired and released within the KCB Oracle functions. Probably, corresponding routines are encoded as C macros.

Shared Latches in eXclusive Mode

- If the latch is being held in **S** mode, the **X** mode waiter will block all further requests.
- This **B**locking is achieved by a special 0x40000000 bit in the latch value. This bit is an indication that some incompatible latch operation is in progress. Only the latch releases are possible.



- Upon release of the latch, the queued processes will be woken up one by one.
- The **X** mode latch gets effectively serialise the shared latch.

Latch Types by Oracle Version

Oracle	Number of latches	PAR	G2C	LNG	UFS	SHARED
7.3.4.0	53	14	2	3	-	-
8.0.6.3	80	21	7	3	-	3
8.1.7.4	152	48	19	4	-	9
9.2.0.8	242	79	37	-	-	19
10.2.0.4	394	117	59	-	4	50
11.1.0.7	502	145	67	-	6	83
11.2.0.3	553	154	72	-	6	93
12.1.0.2	770	241	120	-	8	164

- Oracle added new latches in every version.

Mutex Interface Routines

- **Oradebug watch** allows for catching Oracle functions that modify mutex.
- **DTrace** reveals the flow of and arguments of functions.
- Oracle uses the following KGX functions for changing the mutex state:

kgxExclusive (.,mutex, AOL)	-get the mutex in **X** mode.
kgxShared(.,mutex, AOL)	-get the mutex in **S**hared mode.
kgxSharedExamine(. . .)	
kgxRelease(.,AOL)	-release the mutex.
kgxExclusive2Shared (. . .)	-downgrade **X** mode to **S**.
kgxIncrement(. . .)	-increment **S** mode counter.
kgxDecrement(. . .)	-decrement **S** mode counter.
. . .	

- Some non-KGX functions, such as **kksLockDelete()** also modify mutexes.

Mutex Structure in Memory

- Oracle does not externalise the mutex structure to the SQL:

```
SQL> oradebug peek 0x3F119B5A8 24
[3F119B5A8, 3F119B5CC) =
  00000016 00000001 0000001D 000015D7 382DA701 03000000 ...
       SID    refcnt     gets    sleeps      idn    op
```

The mutex structure contains:

- An atomically modified value that consists of (Note 1298015.1):
  - **Holding SID**. The top 4 bytes contain the SID of the session currently exclusively holding or modifying the mutex.
  - **Reference Count**. The lower 4 bytes represent the number of sessions currently holding the mutex in shared mode (or is in-flux).
- GETS – number of times the mutex was requested.
- SLEEPS – number of times sessions slept for the mutex.
- IDN – mutex identifier. Not unique.
- OP – current mutex operation.

Mutex Value in S and X Modes

- The 8-byte mutex value is changed using atomic CAS instructions (see MOS Notes 727400.1, 1310764.1, 1298015.1).
- **S** mode:
  - Allows the mutex to be held by several sessions simultaneously.
  - **0x00000000|Reference Count** represents the number of sessions holding the mutex.
  - **oper SHRD** in state object dumps.
- **X** mode:
  - Is incompatible with all other modes.
  - Only one session can hold the mutex in exclusive mode.
  - **Holding SID | 0x00000000**
  - **oper EXCL** in object state dumps.
- **LONG_EXCL** is a variant of **X** mode that is used by **Cursor Pin** mutexes.

Mutex Examine Mode

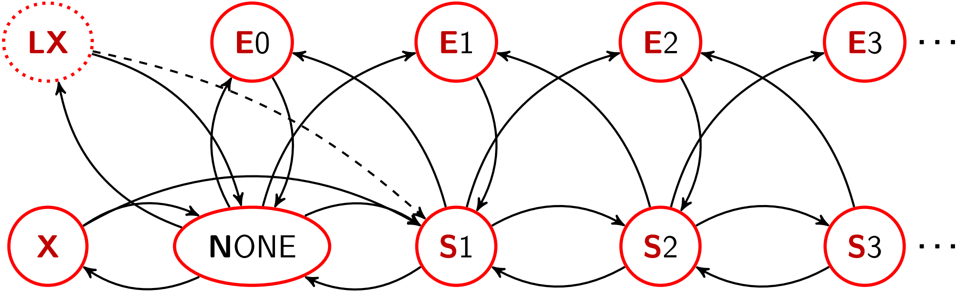
- Examine mode indicates that the mutex is in a transient state. It is neither **X**, nor **S**:

kgxExamine()	- set **E**xamine mode
kgxEndExamine()	- clear the Examine mode
kgxIncrementExamine()	- increment **RefCnt** and set **E** mode
kgxDecrementExamine()	- decrement **RefCnt** and set **E** mode

- **Holding SID|Reference Count**
- In **E** mode, the upper bytes of the mutex value are nonzero and are equal to the holder SID. The lower bytes are also nonzero and represent the number of sessions simultaneously holding the mutex in **S** mode.
- The session can acquire the mutex in **E** mode or upgrade it to **E** mode even if there are other sessions holding the mutex in **S** mode.
- No other session can change the mutex at that time.

Mutex Operations Diagram

```
gdb $ORACLE_HOME/bin/oracle 6441
(gdb) set $op=(&'kgxOpcodeName.0')
(gdb) x/s *($op++)
0x107a538c <_2__STRING.93.0>:    "NONE"
0x10f2c348 <_2__STRING.1.0>:     "GET_SHRD"
0x10e961c4 <_2__STRING.2.0>:     "SHRD"
...  Not all the 19 operations are used by each mutex type
```



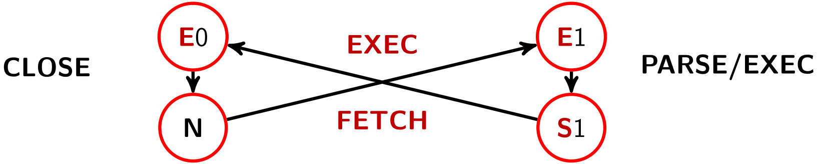
Mutex Types can be Found in Systemstate Dumps

id:	Mutex type:	Type of object it protects:
9	kxs Replay Context	RAT
8	hash table	parent cursor
7	Cursor Pin	child cursor
6	Cursor Parent	...
5	Cursor Stat	...
4	Library Cache	Library cache
3	HT bucket mutex (kdlwl ht)	SecureFiles
2	SHT bucket mutex	...
1	HT bucket mutex	...
0	FSO mutex	...

- **Cursor Pin** mutexes act as pin counters for child cursors.
- **Cursor Parent** and **hash table** mutexes protect parent cursors during parsing and reloading.
- **Library cache** cursor mutexes (idn=*hash value*) protect KGL locks.
- **Library cache** bucket mutexes (idn$\leq$131072) protect static hash structures of the Library Cache.

Cursor Pin Mutex Operations

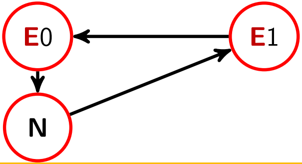
- **Cursor Pin** mutex pins the cursor for parsing and execution:



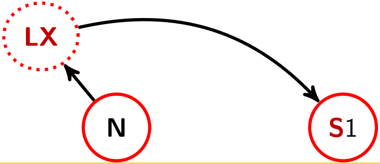
kgxSharedExamine	0000000000	SHRD	->	7d00000001	SHRD_EXAM
kgxEndExamine	7d00000001	SHRD_EXAM	->	0000000001	SHRD
kgxDecrementExamine	0000000001	SHRD	->	7d00000000	DECR_EXAM
kgxEndExamine	7d00000000	DECR_EXAM	->	0000000000	SHRD

- E**X**clusive mutex mode protects the cursor during hard parse:

Cursor invalidation:

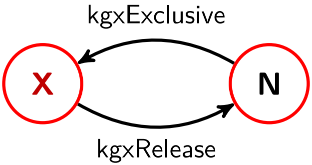


Hard parse:

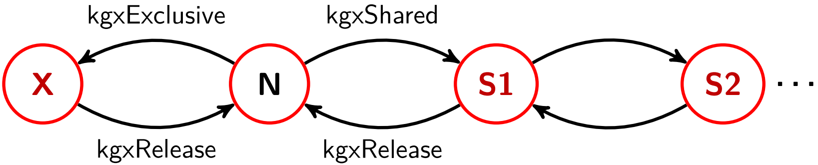


Other Mutexes

- **Library Cache** and **kxs Replay Context** mutexes use **X** mode only, similar to an exclusive latch:



- **Hash table** and **cursor parent** mutexes use **X** and **S** modes:



- There is no blocking of **S** gets by **X** gets.

Latch and mutex spins and waits

Latch Waits in the Oracle Wait Interface

- Oracle registers a latch wait event when the process has failed to acquire the latch after the spinning and goes to sleep.
- There is one general **latch free** and 47 specific **latch:**... wait events.

Wait Event Name	Parameter1	Parameter2	Parameter3
latch free	**address**	**number**	**tries**
latch: cache buffers chains	**address**	**number**	**tries**
latch: enqueue hash chains	**address**	**number**	**tries**
latch: shared pool	**address**	**number**	**tries**
...			
latch: virtual circuit queues	**address**	**number**	**tries**

- Latch waits also are instrumented in the **x$ksupr** (i.e., **v$process**) fixed table:
  - **ksllalaq** – the address of the latch that process is acquiring.
  - **ksllawat** – latch being waited for. This is the **v$process.latchwait**.
  - **ksllawhy** – **why** code for the latch is being waited for.
  - **ksllawere** – **where** for the latch being waited for.

Artificially Busy Latch

- To explore latches spins and waits, I wrote a testcase in which the latch was held for a long period of time.
- In the first session, the latch was artificially acquired and held for 50 s:

```
SQL> oradebug call kslgetl 0x6000AF48 1 100 256
SQL> host sleep 50
SQL> oradebug call kslfre 0x6000AF48
```

- Next, DTrace traced the spin and wait of the attempt to acquire the same latch in the second session:

```
SQL> host /usr/sbin/dtrace -s latch_trace.d -p &spid 0x0x6000AF48 &
SQL> oradebug call kslgetl 0x6000AF48 1 101 255
  kslgetl(0x6000ACC8,1,2,3)  ...
```

- In previous versions of Oracle, the latches were spun via repeated calls of a special routine. *This is no longer the case in Oracle 12c.*
- Recently, Frits Hoogland demonstrated how to find the spinning loop in Oracle 12c using **gdb**.

DTracing the 12.1.0.2 Exclusive Latch Wait

```
kslgetl(0x6000ACC8,1,2,3)           - 0. KSL GET exclusive Latch# 4
 Atomic Operation at kslgetl:9d     - 1. immediate latch get
  kslges(0x6000ACC8, ...)           - 2. wait get of exclusive latch
   spinning ...                     - 3.   20000 cycles
    kslwlmod(...)                   - 4. KSL Wait List MODification
  Atomic Operation at kslges:4bb    - 5.    atomic get
      semop(47,...)                 - 6. wait until posted
```

- By default, the exclusive latch spin have 20,000 cycles.
- Only the first and last reads are atomic; the other reads poll the CPU cache.
- Since Oracle 9.2, all latches of the default class 0 have used the **latch wait posting** without any timeout.
- The static **_latch_classes** and **_latch_class**_x parameters determine the wait and spin of an exclusive latch.
- The **_spin_count** parameter is effectively static for exclusive latches:
  - Its dynamic change does not affect the exclusive latches.
  - Its nondefault value changes the **_latch_class**_x values upon restart.

The Shared Latch Wait

- In contrast to the exclusive latches, the spin of a shared latch can be tuned dynamically.
- The default value of the **_spin_count** parameter is **2,000**.

```
ksl_get_shared_latch(...)                 - 1. shared latch get
 Atomic op. at ksl_get_shared_latch:ac - 2. immediate latch get
  kslgess(0x6000AEA8, ...)                - 3. wait get of shared latch
   spinning ...                           - 4.   2000 cycles
    kslwlmod(...)                         - 5. KSL Wait List MODificati
      semop(16777235,...)                 - 6. wait until posted
```

- Unlike the previous versions, in Oracle 12c:
  - The acquisition of the shared latch spins only once before the sleep.
  - The **S** mode get of the shared latch also spins.

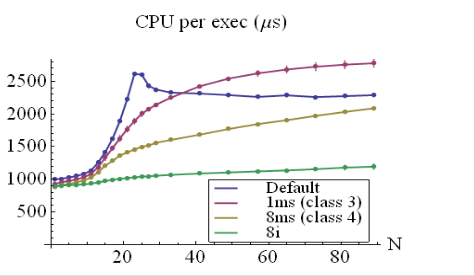
Latch Classes

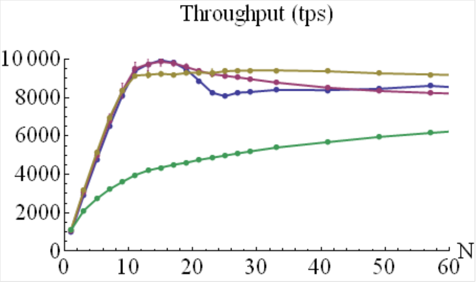
- Oracle defines 8 different latch classes with different spin and wait policies. By default, all the latches except the **process allocation** latch belong to class 0.
- The latch can be assigned to the class by its number. For example:

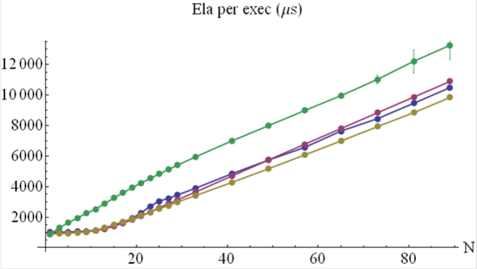
```
 "_latch_classes"='4:6'   "_latch_class_6"='100 1 0 1000 2000'
sql>select * from x$ksllclass;
INDX    SPIN YIELD WAITTIME SLEEP0 SLEEP1 SLEEP2 SLEEP3 SLEEP4 ...
   0  20000     0        1   8000   8000   8000   8000   8000 ...
...6    100     1        0   1000   2000   4000   4000   4000 ...
```

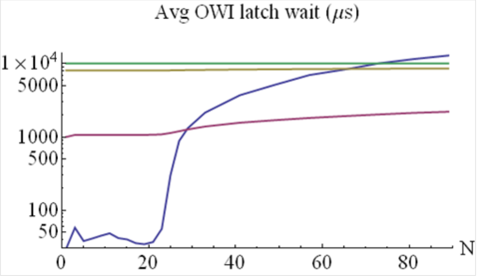
```
...  kslges(0x6000ACC8, ...)                - 2. wait get of exclusive
   spinning ...                             - 3.   100 cycles
    yield()                                 -      Yield 1
   spinning ...                             - 4.   100 cycles
    pollsys(...,timeout=1000 ns,...)        - 5. Sleep 1
   spinning ...                             - 6.   100 cycles
    yield()                                 -      Yield 1
   spinning ...                             - 7.   100 cycles ...
    pollsys(...,timeout=2 ms,...)           - 8. Sleep 2 ...
```

The Performance of the Latch Classes









Row cache objects latch contention testcase on ODA (24 SMT)

New in 12c: Latch Wait List Priority

- After an unsuccessfull spin the process links itself to the end of latch wait list and goes to sleep.
- However, if this is not the first sleep during the long wait for the latch, then the process will get a priority.
- Next **kslfre()** scans the list and posts the process having the priority.
- **_latch_wait_list_pri_sleep_secs** parameter determines the time in seconds to sleep on latch wait list until getting the priority.
- According queuing theory, this functionality does not affect the overall latch performance.
- However, this bounds a heavy-tailed latch waits distribution.

Mutexes in the Oracle Wait Interface

Wait Event Name	Parameter1	Parameter2	Parameter3
cursor: mutex X	idn	value	where
cursor: mutex S	idn	value	where
cursor: pin X	idn	value	where
cursor: pin S	idn	value	where
cursor: pin S wait on X	idn	value	where
library cache: mutex X	idn	value	where
library cache: mutex S	idn	value	where
SecureFile mutex	?	?	?

- Consistency matrix of the mutex modes:

Get \ Held	**S**	**X**,**LX**	**E**
S	—	*mutex type* S	?
X	*mutex type* X	*mutex type* X	*mutex type* X
E	—	*mutex type* S wait on X	*mutex type* S

- We will focus on the **library cache:**... and **cursor: pin**... waits.

Parameters of the Mutex Wait Events

- The parameters of the mutex waits:
  - P1=**idn**. A mutex identifier. This is the hash value of the library cache object protected by mutex or the hash bucket number.
  - P2=**value** = **Blocking SID|Shared refs**. This is the mutex value at the beginning of the wait. It contains the SID of the blocker and the number of shared references.
  - P3=**where** = **Location ID|0**. The top 4 bytes contain location in code where the session is waiting for the mutex.
- Useful scripts from my blog:
  - **mutex_waits.sql –** provides the details regarding the sessions currently waiting for mutexes.
  - **mutex_ash_waits.sql –** provides details regarding the mutex wait history from ASH.

A x$mutex_sleep_history is not a Circular Buffer

- In my mutex contention testcase with two sessions, the fixed table **x$mutex_sleep_history** contains only 3 rows:

```
SQL> select SLEEP_TIMESTAMP,MUTEX_ADDR,MUTEX_IDENTIFIER,MUTEX_TYPE,
       GETS,SLEEPS, REQUESTING_SESSION,BLOCKING_SESSION,
       LOCATION,MUTEX_VALUE from x$mutex_sleep_history;
```

SLEEP TIMESTMP	MUTEX ADDR	MUTEX IDN	MUTEX TYPE	GETS	SLEEPS	REQ	BLK	LOCATION	VALUE
06:19:23. 465130	2CDA1E80	3222383532	Cursor Pin	2973608	1256	136	0	kksfbc [KKSCHLFSP2]	00
06:19:25. 145753	2CDA1E80	3222383532	Cursor Pin	3728317	1552	136	15	kksLockDelete [KKSCHLPIN6]	000F0001
06:19:25. 200443	2CDA1E80	3222383532	Cursor Pin	3742424	1559	15	136	kksLockDelete [KKSCHLPIN6]	00880002

- This fixed table is an array in the SGA hashed by MUTEX_ADDR and BLOCKING_SESSION.
- The row corresponding to the next sleep for the same mutex and blocking session replaces the row for the previous sleep.
- This is the only place where the MUTEX_ADDR is externalised.

x$mutex_sleep and Mutex Locations

- A mutex **location_id** is similar to a latch **where** parameter.
- This is a place in the Oracle code from which the mutex has been requested.
- The location name can be seen in the **x$mutex_sleep** fixed table summarises the sleep statistics of mutexes:
  - LOCATION and LOCATION_ID – the name and ID of the code location where the wait occurs.
  - SLEEPS –number of sleeps for this MUTEX_TYPE and LOCATION.
  - WAIT_TIME – cumulative time (in microseconds) slept at this LOCATION.
- Unlike what it does for the latch, Oracle does not externalise the complete list of the mutex sleep locations.

The Artificially Busy Mutex

- To explore the mutex waits, I will need a testcase that will hold the mutex for a long time.
- AOL and mutex acquisition are too complex for the **oradebug call**.
- *The **oradebug poke** no longer works in Oracle 12c.*
- I will simulate the busy mutex by directly modifying its value from the inside of the Solaris kernel via DTrace:

```
SQL>host dtrace -s mutex_waits.d -p <spid> 0x870d41f0 0x100000001 &
...
   *RefCount= $2;
   copyout(RefCount,$1,8);
...
SQL>oradebug peek  0x870d41f0 8
[0870D41F0, 0870D41F8) = 00000001 00000001
```

- This looks exactly like the SID 1 is holding the mutex in **E** mode.
- The script can simulate the busy mutex in **E**, **S**, and **X** modes.

DTracing the 12.1.0.2 "Library Cache Mutex X" Wait

- The testcase holds a **library cache** mutex corresponding to the PL/SQL procedure **demo_proc** in **X** mode for 50 s.
- One second later, the session tries to execute the **demo_proc** and waits for the **library cache: mutex X** event for 49 s:

```
SQL> exec demo_proc()
...kgxExclusive(.,mutex=0x8941B6E0,aol=0x89065380)
 spinning ...                              - 1.    255 cycles
   yield()                                 - 2.
 spinning ...                              - 3.    256 cycles
   yield()                                 - 4.
 spinning ...                              - 5.    256 cycles
  semsys(...,timeout=10 ms,...)        - 6.
 spinning ...                              - 7.    256 cycles
  semsys(...,timeout=10 ms,...)        - 8. ...
```

- First, the session spins and yields the CPU to other processes twice.
- Then, the session repeatedly spins and sleeps for 10 ms.

The Mutex Waits Schemes Introduced in Oracle 11.2

Event	Waits	%Time -outs	Total Wait Time (s)	Avg wait (ms)	Waits /txn	% DB time
library cache: mutex X	1		49	49142.83	0.07	97.50
db file sequential read	88		0	0.04	6.29	0.01

- **Patch 10411618. Enhancement to add different "mutex wait schemes".**
- Allows 1 of 3 concurrency wait schemes and introduces 3 parameters to control the mutex waits:
  - **_mutex_wait_scheme** – which wait scheme to use.
    0 – Always YIELD.
    1 – Always SLEEP for **_mutex_wait_time**.
    2 – Exponential backoff up to **_mutex_wait_time**.
  - **_mutex_spin_count** – the number of spins. The default is 255.
  - **_mutex_wait_time** – the sleep timeout. The default is 1.
- The default scheme is 2.
- It consumes much less CPU than the aggressive mutex waits in Oracle 10g.

Exponential Backoff in the Mutex Wait Scheme 2

- Surprisingly, **there is no exponential backoff by default**. The session repeatedly sleeps for 1 cs.
- To observe the exponential timeouts, one should increase the **_mutex_wait_time** parameter (hereafter we will omit the spins):

```
SQL> alter system set "_mutex_wait_time"=30;
SQL> SQL> exec demo_proc()
kgxExclusive(.,mutex=0x880CDB28,aol=0x8960DA68)
  yield() call repeated 2 times
  semsys()   timeout=10 ms    call repeated 2 times
  semsys()   timeout=30 ms    call repeated 2 times
  semsys()   timeout=70 ms    call repeated 2 times
  semsys()   timeout=150 ms
  semsys()   timeout=230 ms
  semsys()   timeout=300 ms    call repeated 160 times ...
```

- The **_mutex_wait_time** parameter value is the maximum sleep time in centiseconds.
- This scheme resembles the latch acquisition algorithm in Oracle 8i.

Sleeps Mutex Wait Scheme 1

- Mutex wait scheme 1 repeatedly requests 1 ms sleep:

```
SQL> alter system set "_mutex_wait_scheme"=1;
SQL> exec demo_proc()
kgxExclusive(...)
  yield()
  pollsys()   timeout=1 ms    call repeated 4392 times
```

- The **_mutex_wait_time** parameter is the sleep timeout in milliseconds:

```
SQL> alter system set "_mutex_wait_time"=30;
SQL> exec demo_proc()
kgxExclusive(...)
  yield()
  pollsys()   timeout=30 ms    call repeated 1574 times
```

- **_mutex_wait_time**=0 results in a mutex wait scheme 0.

Classic Mutex Wait Scheme 0

Event	Waits	Total Wait Time (sec)	Wait Avg(ms)	% DB time	Wait Class
library cache: mutex X	7	49.1	7018.72	97.6	Concurrency
DB CPU		6.5		13.0	

- Differs from the aggressive mutex waits in Oracle 10g by 1 ms sleep after 99 yields:

```
SQL> exec demo_proc()
kgxExclusive(...)
  yield() call repeated 99 times
  pollsys()   timeout=1 ms
  yield() call repeated 99 times
  pollsys()   timeout=1 ms
  yield() call repeated 99 times
  pollsys()   timeout=1 ms  ...
```

- This 1 ms sleep significantly reduces the mutex CPU consumption and increases the system robustness.

Yield and Sleep Parameters of Mutex Wait Scheme 0

NAME	VALUE	DESCRIPTION
_wait_yield_mode	yield	Wait Yield – Mode
_wait_yield_hp_mode	yield	Wait Yield – High Priority Mode
_wait_yield_sleep_time_msecs	1	Wait Yield – Sleep Time (milliseconds)
_wait_yield_sleep_freq	100	Wait Yield – Sleep Frequency
_wait_yield_yield_freq	20	Wait Yield – Yield Frequency

- With the default value **_wait_yield_mode**=yield mode, the Oracle process first yields the CPU, then sleeps:

```
alter system set "_mutex_wait_scheme"=0 "_wait_yield_mode"='yield'
      "_wait_yield_sleep_time_msecs"=2 "_wait_yield_sleep_freq"=3;
... kgxSharedExamine()
  yield() call repeated 2 times
  pollsys()   timeout=2 ms
  yield() call repeated 2 times
  pollsys()   timeout=2 ms
  ...
```

_mutex_wait_scheme 0: Sleep Wait Mode

- In the complementary **_wait_yield_mode**=sleep, the Oracle process first sleeps, then yields:

```
sql>alter system set "_wait_yield_mode"='sleep';
sql> exec demo_proc()
kgxExclusive(...)
  pollsys()    timeout=1 ms     call repeated 19 times
  yield()
  pollsys()    timeout=1 ms     call repeated 19 times
  yield()...
```

- The **yield** mutex wait mode may cause CPU starvation when Oracle processes run at different priorities. The higher priority processes yielding the CPU are not preempted by the lower priority processes.
- The **_high_priority_processes** parameter lists the RT priority processes.
- The **_wait_yield_hp_mode** parameter allows us to specify the **sleep** wait mode for the high-priority processes only.

Flexibility of Mutex Wait Scheme 0

- The **_mutex_wait_scheme**=0 is very flexible. For example, I can simulate the aggressive mutex wait behaviour of Oracle 10g:

```
sql>alter system set
"_wait_yield_mode"='yield' "_wait_yield_sleep_time_msecs"=0;
sql> exec demo_proc()
kgxExclusive(...)
...
    yield() call repeated 15332138 times
```

- Alternatively, I can simulate a "pure sleep" wait, which is used by non-exclusive mutex gets:

```
SQL>alter system set
"_wait_yield_sleep_time_msecs"=1 "_wait_yield_sleep_freq"=0;
...
   pollsys()   timeout=1 ms    call repeated 4393 times
```

- Setting the **_mutex_wait_scheme** parameter to a value greater than 2 results in scheme 0 in Oracle 12c.

Cursor Pin Mutex Waits

- Experiments demonstrated that only the e**X**clusive mutex gets are affected by the **_mutex_wait_scheme** parameter in Oracle 12c.

In 12c the mutex wait schemes are only applicable to **library cache: mutex X**, **cursor: pin X**, and **cursor: mutex X** waits.

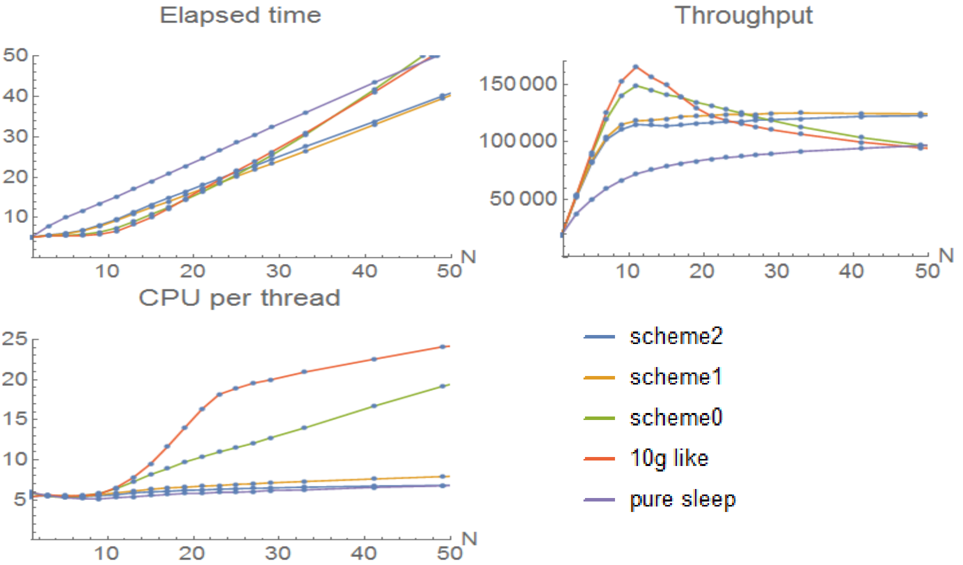
- The examine and shared mutex waits use the "pure sleep" wait scheme without any **yield**'s:

```
SQL> select 1 from dual where 1=2;
kgxSharedExamine(...)
  pollsys()   timeout=1 ms    call repeated 4449 times
```

- The sleep duration is defined by the **_mutex_wait_time**.

Cursor: pin S, **cursor: pin S wait on X**, and **cursor: mutex S** contentions no more tunable by the **_mutex_wait_scheme** parameter.

Comparison of Mutex Wait Schemes



Library cache: mutex X testcase on 12 Cores (24 SMT) X2-2 Exadata.

Comparison of Mutex Wait Schemes

- Default scheme 2 is well balanced in all concurrency regions.
- Scheme 1 should be used when the system is constrained by the CPU.
- Previous Oracle 10.2–11.1 mutex wait algorithm:
  - Had the fastest performance in medium concurrency workloads.
  - Throughput fell if the number of contending threads exceeded the number of CPU cores.
  - CPU consumption increased rapidly beyond this point.
  - This excessive CPU consumption starved the CPUs and impacted the other workloads.
- Scheme 0 has a throughput similar to the Oracle 10g scheme in the medium concurrency region.
- The "pure sleep" wait scheme results in very low CPU consumption, but it has the longest elapsed time and the worst throughput.
- All the contemporary Oracle 12c mutex schemes consume less CPU than those of versions 10.2 and 11.1.

Latch and mutex contention

Oracle Spinlock Contention

- Contention arises when the latch or mutex is requested by several sessions simultaneously.
- Contention is the consequence of over-utilisation or abnormally long holding times for the spinlock. To distinguish between these scenarios, one should investigate the corresponding latch and mutex statistics.
- In most cases, the root cause of latch and mutex contention is a bug inside an application or within Oracle.
- The techniques used to treat spinlock contention in Oracle 12c include:
  - Flexible mutex wait schemes.
  - Latch classes.
  - Tuning of the **spin count**.
  - **Cloning** of hot library cache objects.

Origins of Latch and Mutex Statistics

- The cumulative counters of latch statistics are externalised in **v$latch_parent**/**v$latch_children**:

Statistics:	When and how it increments:
GETS	++ *after* the **wait** mode latch get
MISSES	++ *after* the get if it was **missed**
SLEEPS	+**number_of_sleeps** during the get
SPIN_GETS	++ if the get was **missed** but no sleep occurred
WAIT_TIME	+**wait_time**" *after* the latch acquisition

- The mutex statistics counters are located inside the mutex structure:

```
SQL> oradebug setmypid
Statement processed.
SQL> oradebug peek 0x87F387B8 24
[087F387B8, 087F387C8) = 00000000 00000000 000C352E 000003B5
                         ^mutex                   ^GETS    ^SLEEPS
```

- By sampling of the spinlock value, we can measure its **U**tilisation.

The Key Queuing Spinlock Properties

- Differential (point-in-time) statistics:

Requests arrival rate:	$\lambda = \frac{\Delta\, gets}{\Delta\, time}$
Miss ratio (PASTA):	$\rho = \frac{\Delta misses}{\Delta gets} \approx U$
Avg. holding time (Little's law):	$S = U/\lambda$
Sleeps rate:	$\omega = \frac{\Delta sleeps}{\Delta time}$
Sleeps ratio:	$\kappa = \frac{\Delta sleeps}{\Delta misses} = \omega/(\lambda U)$
Wait time per second:	$W = \frac{\Delta wait\_time}{\Delta time}$
Mutex spin inefficiency:	$k = \kappa/\Big(\rho(1+\rho\kappa)\Big)$

- The average holding time S is the most important aspect of tuning.
- For more information on the mathematics of exclusive latches and mutexes, see my blog.
- To my knowledge, there is no mathematical theory of the shared spinlocks.

Average Latch and Mutex Holding Times

- **latch_stats_11g.sql** – measures and computes statistics for a given latch address.
- **mutex_stats.sql** – measures and computes statistics for a given mutex address *(doesn't work yet in 12c due to bug 19363432)*
- Typical no-contention values for latch and mutex holding time S in exclusive mode on some platforms (us):

	library cache mutex	**session allocation** latch	mutex **spin time**
Sparc T5-8	0.3-3	2-5	0.7
IBM P795	0.3-2	2-3	0.9
Exadata X2-2	0.3-5	5-10	1.8
Sparc T2000	2.5-12	10-15	8.7

- These microsecond intervals are 10,000 times less than those that occur in the 1 centisecond duration of mutex sleep.

Time Microscope Idea

Reality:	**1,000,000X** Zoom:
1 us	1 s
1 ms	17 min
1 s	11.5 days
Light speed (300000 km/s)	Sonic speed (300 m/s)
Mars rocket (11 km/s)	Garden snail (11 mm/s)
CPU tick (2 GHz)	0.0005 s
Normal **library cache** mutex holding time	$\approx 0.3 - 5$ s
Max spin time for mutex (255 spins)	≈ 1 s
Exclusive latch holding time	$\approx 10 - 20$ s
Max spin time for exclusive latch	≈ 1 min
Min interval between mutex gets	$\approx 1 - 2$ s
OS context switch time (10 us – 1ms)	10 s – 17 min

Mutex and Latch Waits Under the Time Microscope

- Mutex wait scheme 2 algorithm:
  - Spin for mutex for 1 s X 3 times.
    - Sleep for 3 hours (1 cs) in hope that the congestion will dissolve.
  - Spin again for 1 s.
    - Sleep again for 3 hours.

  etc.
- The sleep duration for this scheme is much longer than a normal mutex correlation time.
- Wait scheme 1 sleeps for 17 min (1 ms) after 1 s spin. This is still 1000 times longer than the typical mutex time.
- Compare these timed mutex sleeps to the post-wait algorithm for the latch:
  - Spin for exclusive latch for 1 min.
  - If the spin is not successful, set to sleep until post.
- According to **v$event_histogram**, the majority of latch waits take less then 1 ms.

Latch and Mutex Contention Diagnostics

Little's law: $U = \lambda S$

- Spinlock contention should be suspected if the latch or mutex wait events are observed in the "**Top 5 Timed Events**" AWR section.
- Contention can be a consequence of:
  - Abormally long holding time $S > 10$ us due to: high version counts, bugs, CPU starvation, and preemption, etc.
  - High spinlock Utilization due to excessive requests.
- Spinlock statistics helps diagnose what actually happens.
- The **latchprofx.sql** script by Tanel Poder reveals **where** the latch contention arises.

Beware of Bugs and Certain v$/x$

- Scans of some **x$** tables may induce the spinlock contention:

Fixed table	Fixed view	Spinlock
x$ktcxb	**v$transaction**	**transaction allocation** latch.
x$ktadm	**v$lock**, **dba_jobs_running**	**DML lock allocation** latch.
x$ksmsp		**shared pool** latch
x$kslltr	**v$latch**	all the parent latches
x$kqlfxpl	**v$sql_plan**	**library cache** mutex
x$kqlob	**v$sql**	**library cache** mutex
x$kqllk	**v$open_cursor**	**library cache** mutex
x$kqlpn		**library cache** mutex

Divide and Conquer the Mutex Contention

- Contention for heavily accessed objects can be divided between multiple copies of the object in the library cache.
- **dbms_shared_pool.markhot()** marks the hot library cache object as a candidate for cloning.
- The **_kgl_hot_object_copies** parameter controls the number of copies:

```
SQL>exec dbms_shared_pool.markhot('SYS','DEMO_PROC',1);
SQL>select kglnaown,kglnaobj,kglobprop from x$kglob
        where bitand(kglhdflg,33555456) != 0;
KGLNAOWN   KGLNAOBJ   KGLOBPROP
---------- ---------- ----------
SYS        DEMO_PROC  HOT
SYS        DEMO_PROC  HOTCOPY1
SYS        DEMO_PROC  HOTCOPY2 ...
```

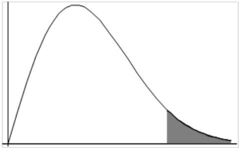
- *“. . . The intention is that in a future release there will be no need to mark copies hot because the RDBMS will be able to detect and mark them itself. . . ”*

Let it Spin

- Latches and mutexes were designed to spin, so let them!
- Longer spinlock holding times may cause the contention. Spinning may mitigate this.
- Default spin counts:
  - **_mutex_spin_count**=255 for mutexes.
  - **_spin_count**=2000 for shared latches.
  - **_latch_class_0**="20000" for exclusive latches.
- Different values may be appropriate for some contention scenarios.

Tuning the Spin Count of Exclusive Spinlocks

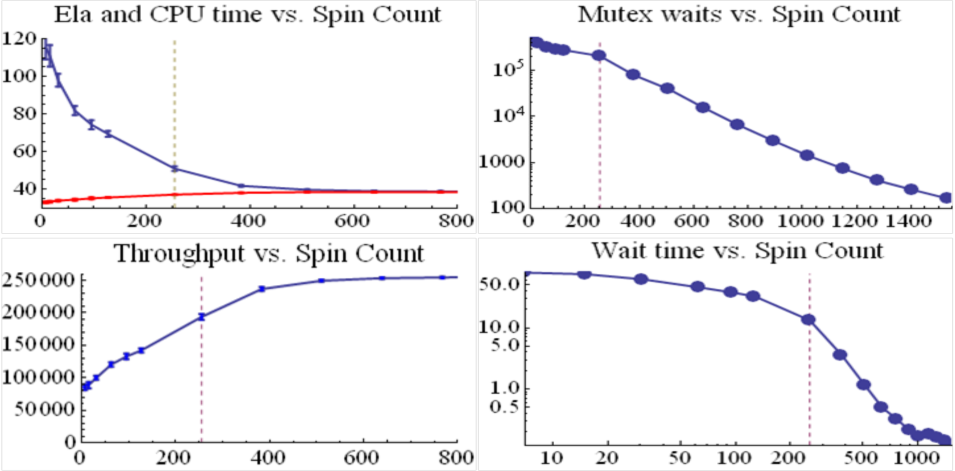
- When the holding time S of the exclusive spinlock is in the range of microseconds and its utilisation is not approaching 100%, then it is mathematically advisable to increase the spin count.
- The belief that CPU time will raise infinitely with increases in spin count is a common myth. In reality, the processes will spin up to an average residual holding time, as follows:



$$\begin{cases} k = \frac{1}{S}\int\limits_{\Delta}^{\infty}(t-\Delta)p(t)\,\mathrm{d}t \\ \Gamma = \frac{\langle t^2\rangle}{2S} - \frac{1}{2S}\int\limits_{\Delta}^{\infty}(t-\Delta)^2 p(t)\,\mathrm{d}t \end{cases}$$

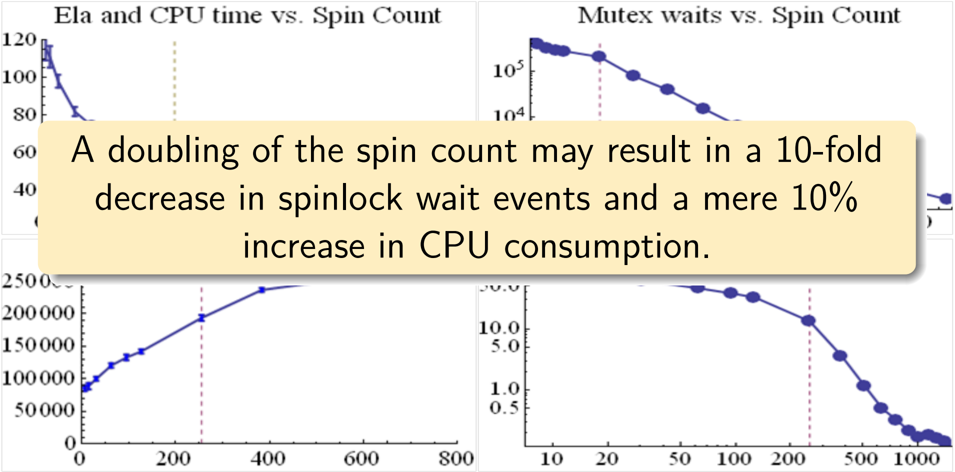
- The spin count tuning probes the holding time distribution.
- Good starting point for tuning is the multiple of default spin count value (*255 or 20K*).
- **Beware of side effects**. You should have enough free CPU.

Spin-Scaling Rule for Exclusive Spinlocks



- If you have CPU resources and the spin is efficient $(k \leq 0.1)$ then, **doubling the spin count will square the spin inefficiency coefficient and add k^{th} part to the CPU consumption.**

Spin-Scaling Rule for Exclusive Spinlocks



A doubling of the spin count may result in a 10-fold decrease in spinlock wait events and a mere 10% increase in CPU consumption.

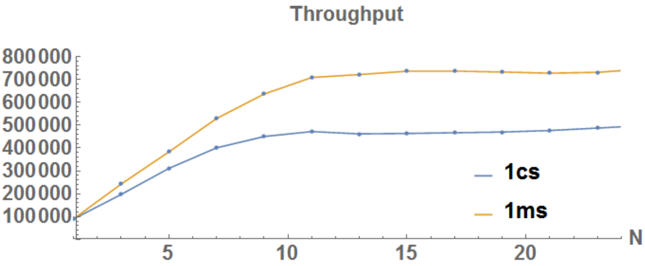
- If you have CPU resources and the spin is efficient $(k \leq 0.1)$ then, **doubling the spin count will square the spin inefficiency coefficient and add k^{th} part to the CPU consumption.**

Spin Count Tuning for the Shared Latches

- Currently, there is no solid mathematical ground for the spin count tuning of the shared latches.
- Empirically, the larger values of the **_spin_count** demonstrate the same spin-scaling exponential behaviour.
- However, for some proportions of the **S** and **X** gets I observed that the decrease of the **_spin_count** increases the throughput.
- Presumably, this is because the **X** mode request serialise the shared latch.
- Hopefully, the spin count adjustment for the shared latch contention may be performed dynamically.

Platform Sleep Granularity

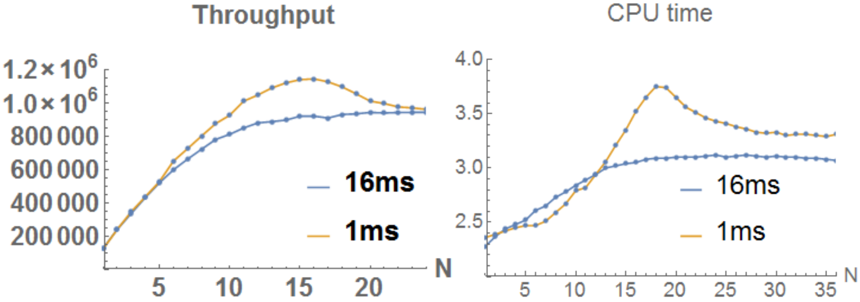
- Oracle 12c requests the 1 ms sleeps for **cursor: pin S** wait event. However, their duration is rounded up to 1 cs on most platforms.
- This results in longer mutex waits and increases the contention:



- Contemporary versions of OEL and Solaris 11.1 support the milliseconds sleep granularity out of the box.
- High resolution sleeps could be enabled on some platforms:
  - On Solaris 10 using static tunable **hires_tick**.
  - On HP-UX 11i V3 via dynamic tunable **hires_timeout_enable**.

Windows is Another World

- Windows default time granularity is 1/64 s=15.625 ms.
- However, it is adjusted to 1 ms by some programs using the **timeBeginPeriod**() API.
- A YouTube video running on the Windows server may resolve the **Cursor: pin S** contention in 12c and boost the performance:



- Oracle GI Cluster Health Monitor (CHM/OS) **osysmond.exe** has a side effect of setting the millisecond time resolution clusterwide.

Q/A?

- Questions?
- Comments?

Acknowledgments

- Thanks to the RDTEX Technical Support Centre Director, S.P. Misiura, for years of encouragement and support of my research.
- Thanks to my brother Aleksey for experiments with Windows.
- Thanks to my colleagues for all of their help.
- Thanks to all our customers who have participated in troubleshooting.
- Thanks to the Oracle development team for the creation of advanced technologies, such as Oracle latches and mutexes.