

Exploring mutexes, the Oracle®RDBMS retrieval spinlocks

Nikolaev A. S.

Andrey.Nikolaev@rdtex.ru, <http://andreynikolaev.wordpress.com>
RDTEX LTD, Protvino, Russia

Proceedings of International Conference on Informatics MEDIAS2012.
Cyprus, Limassol, May 7–14, 2012. ISBN 978-5-88835-023-2

Spinlocks are widely used in database engines for processes synchronization. KGX mutexes is new retrieval spinlocks appeared in contemporary Oracle® versions for submicrosecond synchronization. The mutex contention is frequently observed in highly concurrent OLTP environments.

This work explores how Oracle mutexes operate, spin, and sleep. It develops predictive mathematical model and discusses parameters and statistics related to mutex performance tuning, as well as results of contention experiments.

Исследование мьютексов, спинблокировок с повторными вызовами в СУБД Oracle®

Николаев А. С.

Протвино, ЗАО РДТЕХ

Спинблокировки широко используются в системах управления базами данных для синхронизации процессов. В современных версиях СУБД Oracle появились KGX мьютексы - новый тип спинблокировок с повторными вызовами ориентированный на синхронизацию в субмикросекундных масштабах. Конкуренция за мьютексы часто наблюдается в высоконагруженных OLTP средах.

В работе исследуются особенности циклирования и ожиданий мьютексов. На основе построенной математической модели обсуждаются параметры и статистики связанные с настройкой производительности мьютексов, а также результаты экспериментов.

1. Introduction

According to Oracle® documentation [1] mutex is: "A mutual exclusion object ... that prevents an object in memory from aging out or from being corrupted ...".

Huge Oracle RDBMS instance contains thousands processes accessing the shared memory. This shared memory named "System Global Area" (SGA) consists of millions cache, metadata and results structures. Simultaneous access to these structures is synchronized by Locks, Latches and KGX Mutexes:

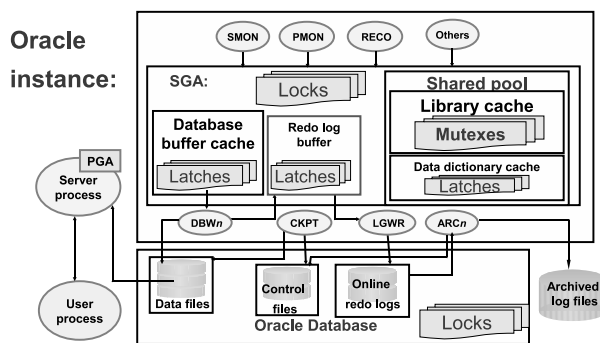


Fig. 1. Oracle® RDBMS architecture

Latches and mutexes are the Oracle realizations of spinlock concept. My previous article [30] explored latches, the traditional Oracle spinlocks known since 1980th. Process requesting the latch spins 20000 cy-

cles polling its location. If unsuccessful, process joins a queue and uses wait-posting to be awoken on latch release.

The goal of this work is to explore the newest Oracle spinlocks — mutexes. The mutexes were introduced in 2006 for synchronization inside Oracle Library Cache. Table 1 compares Oracle internal synchronization mechanisms.

Wikipedia defines the spinlock as "...a lock where the thread simply waits in a loop ("spins") repeatedly checking until the lock becomes available. As the thread remains active but isn't performing a useful task, the use of such a lock is a kind of busy waiting".

Use of spinlocks for multiprocessor synchronization was first introduced by Edsger Dijkstra in [2]. Since that time, the mutual exclusion algorithms were significantly advanced. Various sophisticated spinlock realizations (TS, TTS, Delay, MCS, Anderson, etc.) were proposed and evaluated. The contemporary review of these algorithms may be found in [3]

Two general spinlock types exist:

System spinlocks that protect critical OS structures. The kernel thread cannot wait or yield the CPU. It must loop until success. Most mathematical models explore this spinlock type. Major metrics to optimize system spinlocks are frequency of atomic operations (or Remote Memory References) and shared bus utilization.

User application spinlocks like Oracle latches and mutexes that protect user level structures. It is more efficient to poll the mutex for several microseconds rather than pre-empt the thread doing 1 millisecond context switch. The metrics to optimize are spinlock acquisition CPU and elapsed times.

In Oracle versions 10.2 to 11.2.0.1 (or, equivalently, from 2006 to 2010) the mutex spun using atomic *"test-and-set"* operations. According to Anderson classification [4] it was a **TS** spinlock. Such spinlocks may induce the Shared Bus saturation and affect performance of other memory operations.

Since version 11.2.0.2 processes poll the mutex location nonatomically and only use atomic instructions to finally acquire it. The contemporary mutex became a **TTS** (*"test-and-test-and-set"*) spinlock.

System spinlocks frequently use more complex structures than TTS. Such algorithms, like famous MCS spinlocks [5] were optimized for 100% utilization. For the current state of system spinlock theory see [6].

If user spinlocks are holding for a long time, for example due to OS preemption, pure spinning becomes ineffective and wastes CPU. To overcome this, after 255 spin cycles the mutex sleeps yielding the processor to other workloads and then retries. The sleep timeouts are determined by mutex *wait scheme*.

Such *spin-sleeping* was first introduced in [7] to achieve balance between CPU time lost by spinning and context switch overhead.

From the queuing theory point of view such systems with repeated attempts are retrial queues. More precisely, in the retrial system the request that finds the server busy upon arrival leaves the service area and joins a retrial group (*orbit*). After some time this request will have a chance to try its luck again. There exists an extensive literature on the retrial queues. See [8, 9] and references therein.

The mutex retrial spin-sleeping algorithm significantly differs from the FIFO spin-blocking used by Oracle latches [30]. The spin-blocking was explored in [10, 11, 12]. Its robustness in contemporary environments was recently investigated in [13].

Historically the mutex contention issues were hard to diagnose and resolve [28]. The mutexes are much less documented then needed and evolve rapidly. Support engineers definitely need more mainstream science support to predict the results of changing mutex parameters. This paper summarizes author's work on the subject. Additional details may be found in my blog [29].

II. Oracle® RDBMS Performance Tuning overview

Before discussing the mutexes, we need some introduction. During the last 33 years, Oracle evolved

	Locks	Latches	Mutexes
Access	Several - Modes	Types and Modes	Operations
Acquisition	FIFO	SIRO spin FIFO wait	SIRO
SMP Atomicity	No	Yes	Yes
Timescale	Milli-seconds	Micro-seconds	SubMicro-seconds
Life cycle	Dynamic	Static	Dynamic

Table 1. Serialization mechanisms in Oracle

from the first one-user SQL database to the most advanced contemporary RDBMS engine. Each version introduced performance and concurrency advances:

- v. 2** (1979): the first commercial SQL RDBMS.
- v. 3** (1983): the first database to support SMP.
- v. 4** (1984): read-consistency, Database Buffer Cache.
- v. 5** (1986): Client-Server, Clustering, Distributed Database, SGA.
- v. 6** (1988): procedural language (PL/SQL), undo/redo, latches.
- v. 7** (1992): Library Cache, Shared SQL, Stored procedures, 64bit.
- v. 8/8i** (1999): Object types, Java, XML.
- v. 9i** (2000): Dynamic SGA, Real Application Clusters.
- v. 10g** (2003): Enterprise Grid Computing, Self-Tuning, **mutexes**.
- v. 11g** (2008): Results Cache, SQL Plan Management, Exadata.
- v. 11gR2** (2011): ...**Mutex wait schemes. Hot object copies.**
- v. 12c** (2012): *Cloud...*

As of now, Oracle is the most widely used SQL RDBMS. In majority of workloads it works perfectly. However, quick search finds more than 100 books devoted to Oracle performance tuning on Amazon [16, 17, 18]. Dozens conferences covered this topic every year. Why Oracle needs such a tuning?

The main reason is complex and variable workloads. Oracle is working in very different environments ranging from huge OLTPs, petabyte OLAPs to hundreds multi-tenant databases running on one server. Every high-end database is unique.

For the ability to work in such diverse conditions Oracle RDBMS has complex internals. To get the most out of hardware we need precise tuning. Working at Support, I cannot underestimate the importance of developers and database administrators education in this field.

In order to diagnose performance problems Oracle instrumented its software. Every Oracle session keeps many statistics counters describing *"what was done"*.

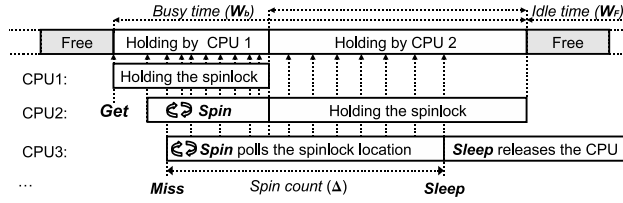


Fig. 2. Oracle mutex workflow.

Oracle Wait Interface (OWI) [18] events describe “why the session waits” and complement the statistics.

Statistics, OWI and data from internal (“fixed”) X\$ tables are used by Oracle diagnostics and visualization tools.

This is the traditional framework of Oracle performance tuning. However, it was not effective enough in spinlocks troubleshooting.

The DTrace.

To observe the mutexes work and short duration events, we need something like stroboscope in physics. Likely, such tool exists in Oracle Solaris™. This is DTrace, Solaris 10 Dynamic Tracing framework [20].

DTrace is event-driven, kernel-based instrumentation that can see and measure all OS activity. It defines **probes** to trap and handlers (**actions**) using dynamically interpreted C-like language. No application changes needed to use DTrace. This is very similar to triggers in database technologies.

DTrace can:

- Catch any event inside Solaris and function call inside Oracle.
- Read and change any address location in-flight.
- Count the mutex spins, trace the mutex waits, perform experiments.
- Measure times and distributions up to microsecond precision.

Unlike standard tracing tools, DTrace works in Solaris kernel. When process entered probe function, the execution went to Solaris kernel and the DTrace filled buffers with the data. Kernel based tracing is more stable and have less overhead then userland. DTrace sees all the system activity and can account the time associated with kernel calls, scheduling, etc.

In the following sections describing Oracle performance tuning are interleaved by mathematical estimations.

III. Mutex spin model

The Oracle mutex workflow schematically visualised in fig. 2. The Oracle process:

- Uses atomic hardware instruction for mutex *Get*.
- If *missed*, process spins by polling mutex location during *spin get*.
- Number of spin cycles is bounded by *spin count*.
- If spin get not succeeded, the process acquiring mutex *sleeps*.

- During the sleep the process may wait for already free mutex.

Oracle counts *Gets* and *Sleeps* and we can measure *Utilization*.

This section introduces the mathematical model used to forecast mutex behaviour. It extends the model used in [10, 30] for general holding time distribution and TTS spinlock concurrency.

Consider a general stream of mutex holding events. The mutex memory location have been changed by sessions at time H_k , $k \in \mathcal{N}$ using atomic instruction. This instruction blocked the shared bus and succeeded only when memory location is free.

After acquisition the session will hold the mutex for time x_k distributed with p.d.f. $p(t)$. I assume that incoming stream is Poisson with rate λ and H_k (and x_k) are generally independent forming renewal process. Furthermore, I assume here the existence of at least second moments for all the distributions.

The mutex acquisition request at time T_m , $m \in \mathcal{N}$ succeeds immediately if it finds the mutex free. Due to *Serve-In-Random-Order* nature of spinlocks, there is no simple relation between m and k .

If the mutex was busy at time T_m :

$$H_k < T_m < H_k + x_k \quad \text{for some } k,$$

then *miss* occurred. According to PASTA property the *miss* probability is equal to mutex utilization ρ .

Missing session will *spin* polling the mutex location up to time Δ determined by `_mutex_spin_count` parameter. The initial “contention free” approximation assumes that no other requests arrive during the spin ($\lambda\Delta \ll 1$) and the session acquires the mutex if it become free while spinning. Therefore the spin succeeds if:

$$T_m + \Delta > H_k + x_k.$$

If the mutex was not released during Δ , the session sleeps.

According to classic considerations of renewal theory [22, 23], incoming requests peek up the holding intervals with p.d.f.:

$$p_h = \frac{1}{S}xp(x), \quad (1)$$

and observes the transformed mutex holding time distribution. Here $S = E(x)$ is the average mutex holding time. The p.d.f. and average of residual holding time is well-known:

$$p_r(x) = \frac{1}{S} \int_0^\infty p(t) dt \quad (2)$$

$$S_r = \frac{1}{2S} E(x^2)$$

The spin time distribution (conditioned on miss) follows the c.d.f. P_r , but has a discontinuity [30] at $t = \Delta$ because the session acquiring latch never spins

more than Δ . The magnitude of this discontinuity is the overall probability that residual mutex holding time will be greater than Δ . Corresponding p.d.f. is:

$$p_{sg}(x) = p_r(x)H(\Delta - x) + Q_r(\Delta)\delta(x - \Delta) \quad (3)$$

Here $H(x)$ and $\delta(x)$ is Heaviside step and bump functions correspondingly.

The spinlock observables

Oracle statistics allows measuring of *spin inefficiency* (or *sleep ratio*) coefficient k . This is the probability do not acquire mutex during the spin. Another crucial quantity is Γ - the average CPU time spent while spinning for the mutex:

$$\begin{cases} k_0 = Q_r(\Delta) = \int_{\Delta}^{\infty} p_r(t) dt = \frac{1}{S} \int_{\Delta}^{\infty} Q(t) dt \\ \Gamma = \int_0^{\infty} t p_{sg}(t) dt = \frac{1}{S} \int_0^{\Delta} dt \int_t^{\infty} Q(z) dz \end{cases} \quad (4)$$

Here subscript 0 denotes "contention free" approximation. Using (2) for distributions with finite dispersion this expressions can be rewritten in two ways [30].

Low spin efficiency region

The first form is suitable for the region of low spin efficiency $\Delta \ll S$:

$$\begin{cases} k_0 = 1 - \frac{\Delta}{S} + \frac{1}{S} \int_0^{\Delta} (\Delta - t) p(t) dt \\ \Gamma = \Delta - \frac{\Delta^2}{2S} + \frac{1}{2S} \int_0^{\Delta} (\Delta - t)^2 p(t) dt \end{cases} \quad (5)$$

From the above expressions it is clear that spin probes the mutex holding time distribution around the origin.

Other parts of mutex holding time p.d.p. impact spin efficiency and CPU consumption only through the average holding time S . This allows to estimate how these quantities depend upon mutex_spin_count (or Δ) change. If process never releases mutex immediately ($p(0) = 0$) then

$$\begin{cases} k = 1 - \frac{\Delta}{S} + O(\Delta^3) \\ \Gamma = \Delta - \frac{\Delta^2}{2S} + O(\Delta^4) \end{cases}$$

For Oracle performance tuning purpose we need to know what will happen if we double the Δ :

In low efficiency region doubling the spin count will double the number of efficient spins and also double the CPU consumption.

High spin efficiency region

In high efficiency region the sleep cuts off the tail of spinlock holding time distribution:

$$\begin{cases} k_0 = \frac{1}{S} \int_{\Delta}^{\infty} (t - \Delta) p(t) dt \\ \Gamma = \frac{E(t^2)}{2S} - \frac{1}{2S} \int_{\Delta}^{\infty} (t - \Delta)^2 p(t) dt = S_r - T_r \end{cases} \quad (6)$$

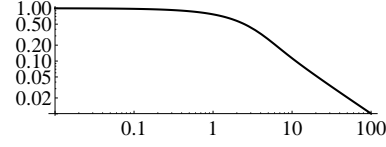


Fig. 3. The concurrency formfactor $F_c(x)$.

here T_r is the residual after-spin holding time. This quantity will be used later.

Oracle normally operates in this region of small sleeps ratio. Here the spin count is greater than number of instructions protected by mutex $\Delta \geq S$. The spin time is bounded by both the "residual holding time" and the spin count:

$$\Gamma < \min(S_r, \Delta)$$

The sleep prevents process from waste CPU for spinning on heavy tail of mutex holding time distribution

Concurrency model

In the real world several processes may spin on different processors concurrently. After the mutex release all these sessions will issue atomic *Test-and-Set* instructions to acquire the mutex. Only one instruction succeeds. What should the other sessions do? This is the principal question for hybrid TTS spinlocks.

The session may either *continue the spin* upto mutex_spin_count or it may proceed to *sleep* immediately. The sleep seems reasonable because the session knows that spinlock just became busy.

For further estimations I will use the second scenario. After the mutex release only one spinning session acquires it according to SIRO discipline, while all other sessions sleep. This is interesting queuing discipline that to my knowledge has not been explored in literature. Its C pseudocode looks like:

```
while(1){ i:=0;
  while(lock<>0 && i<spin_count) i:=i+1;
  if(Test_and_Set(lock))return SUCCESS;
  Sleep();
}
```

The time just after the mutex release is the *Markov regeneration point*. All the spinning behavior after this time is independent of previous history.

Consider mutex holding interval of length x containing at least one (tagged) incoming request for mutex from Poisson stream with rate λ . The conditional probability that this interval will contain exactly n incoming requests is:

$$\frac{1}{1 - e^{-\lambda x}} \frac{(\lambda x)^n}{n!} e^{-\lambda x}, \quad n \geq 1.$$

The session will acquire mutex at these conditions with probability $1/n$. Overall probability for tagged session to acquire mutex is:

$$F_c(x) = \frac{1}{e^{\lambda x} - 1} \sum_{n=1}^{\infty} \frac{(\lambda x)^n}{n!} = -\frac{\text{Ein}(-\lambda x)}{e^{\lambda x} - 1} \quad (7)$$

Here $\text{Ein}(z) = \int_0^z (1 - e^{-y}) \frac{dy}{y}$ is the *Entire Exponential integral* [24].

The *concurrency formfactor* $F_c(x)$ is a smooth monotonically decreasing function with asymptotics $F_c(x) = 1 - x/2 + O(x^2)$ around 0 and $F_c(x) \approx 1/x + O(1/x^2)$, $x \rightarrow \infty$. It may be efficiently approximated by rational functions [25]. Fig. 3 shows its Log-Log plot.

Normally mutex operates in region $\lambda x \ll 1$ and value of formfactor F_c is very close to 1.

According to (1) the probability that missing request observe the holding interval from x to $x + dx$, its residual holding time will be from t to $t + dt$, $t \leq x$ and it will concurrently acquire mutex on release is:

$$dP = \frac{1}{S} p(x) F_c(\lambda \min(x, \Delta)) dx dt$$

Therefore, the *spin inefficiency* or probability not to acquire mutex by spin will be:

$$k = 1 - \frac{1}{S} \int_0^{\Delta} dt \int_t^{\infty} p(x) F_c(\lambda \min(x, \Delta)) dx \quad (8)$$

Changing the integrations order we have:

$$k = 1 - \frac{1}{S} \int_0^{\infty} \min(x, \Delta) p(x) F_c(\lambda \min(x, \Delta)) dx \quad (9)$$

Comparing with (6) we can outline the contention contribution:

$$k = k_0 + \frac{1}{S} \int_0^{\Delta} Q(x) \frac{\partial}{\partial x} (x(1 - F_c(\lambda x))) dx \quad (10)$$

Using the formfactor asymptotics in low contention region $\lambda \Delta \ll 1$ and Little law $\rho = \lambda S$ we can estimate how the spin efficiency depends on mutex utilization:

$$k = k_0 + \frac{\rho}{S^2} \int_0^{\Delta} x Q(x) dx + o((\lambda \Delta)^2)$$

Data of mutex contention experiments (Fig. 4) roughly agree with this linear approximation.

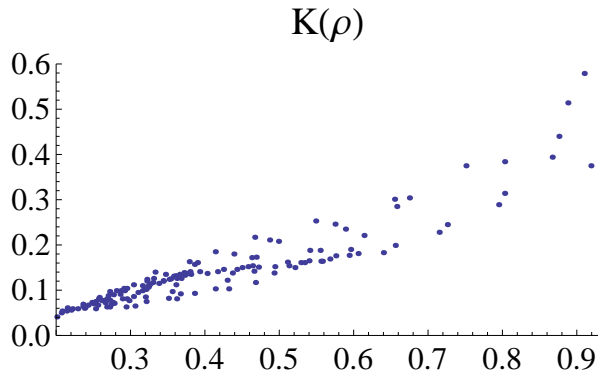


Fig. 4. $k(\rho)$.

IV. How Oracle requests the mutex

This section discusses Oracle mutex internals beyond the documentation. In order to explore mutexes we need reproducible testcases.

Each time the Oracle session executes SQL operator, it needs to *pin* the cursor in library cache using mutex. "True" mutex contention arises when the same SQL operator executes concurrently at high frequency. Therefore, simplest testcase for "Cursor: pin S" contention should look like:

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

The script uses PL/SQL loop to execute fast SQL operator one million times. Pure "Cursor: pin S" mutex contention arises when I execute this script by several simultaneous concurrent sessions.

It is worth to note that **session_cached_cursors** parameter value must be nonzero to avoid soft parses. Otherwise, we will see contention for "Library Cache" and "hash table" mutexes also. Indeed it is enough to disable session cursor cache and add dozen versions of the SQL to induce the "Cursor: mutex S" contention.

Similarly, "Library cache: mutex X" contention arises when anonymous PL/SQL block executes concurrently at high frequency.

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

Many other mutex contention scenarios possible. See blog [29].

Table 2 describes types of mutexes in contemporary Oracle. The "Cursor Pin" mutexes act as pin counters for library cache objects (e.g. child cursors) to prevent their aging out of shared pool. "Library cache" cursor and bucket mutexes protect KGL locks and static library cache hash structures. The "Cursor Parent" and "hash table" mutexes protect parent cursors during parsing and reloading.

The mutex address can be obtained from **x\$mutex_sleep_history** Oracle table. Such "fixed" tables externalize internal Oracle structures to SQL. Due to dynamic nature of mutexes, Oracle does not have any fixed table like **v\$latch** containing data about all mutexes.

According to Oracle documentation, this table is circular buffer containing data about latest mutex waits. However, my experiments demonstrated that it is actually hash array in SGA. The hash key of this array is likely to depend on mutex address and the ID of blocking session. Row for each next sleep for the same mutex and blocking session replaces the row for previous sleep.

	get S	get X,LX	get E
Held S	-	<i>mutex S</i>	?
X	<i>mutex X</i>	<i>mutex X</i>	<i>mutex X</i>
E	-	<i>mutex S wait on X</i>	<i>mutex S</i>

Table 3. Mutex waits in Oracle Wait Interface

When session is waiting for mutex, it registers the wait in Oracle Wait Interface [18]. Most frequently observed waits are named "**cursor: pin S**", "**cursor: pin S wait on X**" and "**library cache: mutex X**". The naming scheme is presented in table 3. Here *mutex* is the name of mutex type.

Experimental setup to explore mutex wait

Unlike the latch, the details of mutex wait were not documented by Oracle. We need explore it using DTrace. To explore the latch in [30], I acquired it directly calling **kslgetl** function. This is not possible for mutex. However, by changing memory I can make mutex "busy" artificially. Oracle **oradebug** utility allows changing of any address inside SGA:

```
SQL>oradebug poke <mutex addr> 8 0x100000001
BEFORE: [3A9371338, 3A9371340) =
                                00000000 00000000
AFTER:  [3A9371338, 3A9371340) =
                                00000001 00000001
```

This looks exactly like session with SID 1 is holding the mutex in **E** mode. I wrote several scripts that simulate a busy mutex in **S**, **X** and **E** modes. In these scripts one session artificially holds mutex for 50s. Another session tries to acquire mutex and "statically" waits for "**cursor: pin S**" event during 49s. DTrace allowed me explore how Oracle actually waits for mutex.

Original Oracle 10g mutex busy wait

Oracle introduced the mutexes in version 10.2.0.2. Running the script against this version I saw that the waiting process consumed one of my CPUs completely. Oracle showed millions microsecond waits that accounted for 3 seconds out of actual 49 second wait. The wait trace looks like:

```
... spin 255 cycles
  yield()
  spin 255 cycles
  yield()
... repeated 1910893 times
```

The session waiting for mutex repeatedly spins 255 times polling the mutex location and then issues **yield()** OS syscall. This syscall just allows other processes to run.

Oracle 10.2-11.1 counts wait time as the time spent off the CPU waiting for other processes. If the system has free CPU power, Oracle thought it was not waiting at all and mutex contention was invisible.

Therefore the old version mutex was "classic" spinlock without sleeps. If the mutex holding time is always small, this algorithm minimizes the elapsed time to acquire mutex. Spinning session acquires mutex immediately after its release.

Such spinlocks are vulnerable to variability of holding time. If sessions hold mutex for a long time, pure spinning wastes CPU. Spinning sessions can aggressively consume all the CPUs and affect the performance by priority inversion and CPU starvation.

Mutex wait with Patch 6904068

If long "cursor: pin S" waits were consistently observed in Oracle 10.2-11.1, then system do not have enough spare CPU for busy waiting. For such a case, Oracle provides the possibility to convert "busy" mutex wait into "standard" sleep. This enhancement was named "*Patch 6904068: High CPU usage when there are "cursor: pin S" waits*". With this patch the mutex wait trace becomes:

```
... spin 255 times
  semsys()   timeout=10 ms
... repeated 4748 times
```

The **semtimedop()** is "normal" OS sleep. The patch significantly decreases CPU consumption by spinning. Its drawback is larger elapsed time to obtain mutex. Ten milliseconds is long wait in Oracle timescale.

One can adjust sleep time with centisecond granularity and even set it to 0 dynamically. In such case the Oracle instance behaves exactly like without the patch. It makes sense to install the patch 6904068 in 10.2-11.1 OLTP environments proactively.

IV. Mutex statistics

Mutex statistics are the tools to diagnose its efficiency. Oracle internally counts the numbers of gets and sleeps for mutex. However, there is no fixed table containing current statistics. The **x\$mutex_sleep_history** shows statistics as they were *at the time of last sleep*. This is not enough.

Hopefully, Oracle provide us the **x\$ksmmem** fixed table. It shows contents of any address inside SGA. The mutex value, its *gets* and *sleeps* can be directly read out from Oracle memory. Repeatedly sampling mutex value we can estimate another key mutex statistics - *Utilization*. The Little's law $U = \lambda S$ allows computing the average mutex holding time S .

Unlike the latches [30], mutex do not count its misses and spin gets. The *miss ratio* ρ should be estimated from PASTA (Poisson Arrivals See Time Averages) property $\rho \approx U$.

Oracle counts only the first mutex get, but all the secondary sleeps. Therefore, the *spin inefficiency* coefficient k differs from experimentally observed *sleep ratio* $\varkappa = \frac{\Delta \text{sleeps}}{\Delta \text{misses}}$.

If sleeps are much longer then mutex correlation time then every *spin-and-sleep* cycle observe indepen-

Description	Definition	Relations
Mutex requests arrival rate	$\lambda = \frac{\Delta_{\text{gets}}}{\Delta_{\text{time}}}$	
Sleeps rate	$\omega = \frac{\Delta_{\text{sleeps}}}{\Delta_{\text{time}}}$	$\omega = \varkappa \rho \lambda$
Miss ratio (PASTA estimation)	$\rho = \frac{\Delta_{\text{misses}}}{\Delta_{\text{gets}}}$	$\rho \approx U_X$
Sleeps ratio	$\varkappa = \frac{\Delta_{\text{sleeps}}}{\Delta_{\text{misses}}}$	$\varkappa = \frac{\omega}{\lambda \rho} = \frac{k}{1 - k\rho}$
Avg. holding time (Little's law)	$S = \frac{U}{\lambda}$	
Mutex spin inefficiency	$k = \frac{\Delta_{\text{sleeps}}}{\Delta_{\text{spins}}}$	$k = \frac{\varkappa}{1 + \varkappa \rho}$

Table 4. Mutex statistics.

dent picture. Due to this independency each sleep has equal probability $k\rho$ and one can estimate:

$$\varkappa = k + (k\rho)k + (k\rho)^2k + \dots = \frac{k}{1 - k\rho}$$

Table 4 summarizes mutex statistics and their relations. Corresponding script **mutex_statistics.sql** to measure mutex statistics is available in [29].

The spin time Δ can be obtained in my experiments by counting the "spin-and-yield" cycles per second. Contemporary Oracle versions can adjust Δ using parameter **_mutex_spin_count**. Therefore, we can compute spin and yield times separately by linear regression.

Typical nocontention values for spin, yield and mutex holding time S in exclusive mode on some platforms are summarized in table 5.

	Library cache	Cursor pin	spin	yield()
Exadata	0.3 – 5 μ s	0.1 – 2 μ s	1.8 μ s	0.7 μ s
Sparc T2	2.5 – 12 μ s	3.2 – 11 μ s	8.7 μ s	9.5 μ s

Table 5. Average mutex spin and **yield()** times.

Compare these microsecond times with default mutex sleep of 10 ms duration. Indeed, the mutex sleep is 10000 times longer than spin.

V. "Mean Value Analysis" of mutex retries

"Mean Value Analysis" (MVA) is an elegant approach for queuing systems invented by M. Reiser, et al. [26]. Recent work [27] discussed the MVA for retrieval queues. Though not applicable directly to non-Markovian mutex, this approach can be useful for estimations.

The important point of the following approximation is replacement of fixed time mutex sleep by exponential memoryless distribution. According to PASTA, request arriving with frequency λ finds mutex busy with probability ρ and goes to orbit (sleeps) for time T with probability $k\rho$.

The waiting time consist of spin and sleep in the orbit times.

$$W = W_s + W_{orb} \quad (11)$$

The process acquires the busy mutex during repeating spins. The total spin time is:

$$W_s = \rho\Gamma + (k\rho)\rho\Gamma + (k\rho)^2\rho\Gamma + \dots = \frac{\rho}{1 - k\rho}\Gamma \quad (12)$$

The request retries from orbit while mutex is busy and idle (Fig. 2).

$$W_{orb} = W_b + W_i$$

In steady state the overall busy mutex wait time is needed to serve all requests currently in system.

$$W_b + W_s = L_{orb}S + L_sS + \rho S_r$$

Here S_r is the residual mutex holding time. According to Little's law:

$$L_{orb} = \lambda W_{orb}, \quad L_b = \lambda W_b, \quad L_s = \lambda W_s, \quad \rho = \lambda S.$$

Therefore:

$$W_b = \rho(S_r + W_{orb}) - (1 - \rho)W_s \quad (13)$$

Flows per second going to and from the orbit should be balanced. For exponential sleep approximation:

$$k\lambda\rho + k\frac{\lambda W_b}{T} = \frac{\lambda W_{orb}}{T}$$

here T is the average time to sleep. One can substitute, in spirit of MVA, the (13) into this expression and estimate the average wait time spent on orbit as:

$$W_{orb} = \frac{k}{1 - k\rho} (\rho(T + S_r) - (1 - \rho)W_s)$$

The overall wait time became:

$$W = \frac{\rho}{1 - k\rho} \left(\frac{1 - k^2\rho}{1 - k\rho} \Gamma + k(T + T_r) \right), \quad (14)$$

where T_r is the residual after-spin mutex holding time that already appeared in (6).

Normally in Oracle 11.2 the spin inefficiency $k \ll 1$ and huge sleep time $T \sim 10^4 \times \{\Gamma, \Delta, T_r\}$ dominates in these formulas and limits the mutex wait performance

$$W \approx \frac{k\rho}{1 - k\rho} (T + T_r).$$

In order to compare this to mutex experimental data it should be noted that, unlike the queuing theory, the Oracle Wait Interface does not treat the first spin as a part of wait[18]. The wait time registered by OWI is $W_o = W - \rho\Gamma$.

Oracle performance tuning uses an "average wait duration" metric from AWR report [1] as a contention

signature. This is the OWI waiting time normalized by the *number of OWI waits*:

$$\bar{w}_o = W_o / k\rho \approx \frac{1}{1 - k\rho} (T + T_r)$$

If this quantity significantly differs from 1cs, it may be a sign of abnormality in mutex utilization or holding time.

Of course the above estimations do not account for OS scheduling and are not applicable when number of active processes exceeds the number of CPUs.

VI. 11.2.0.2.2 Mutex waits diversity

Since April 2011 the latest Oracle versions use completely new concept of mutex waits.

My Oracle Support site [15] described this in note *Patch 10411618 Enhancement to add different "Mutex" wait schemes*. The enhancement allows one of three 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 upto `_mutex_wait_time`.
- `_mutex_spin_count`** — the number of times to spin. Default value is 255.
- `_mutex_wait_time`** — sleep timeout depending on scheme. Default is 1.

The note also mentioned that this fix effectively supersedes the patch 6904068 described above.

The SLEEPS. Mutex wait scheme 1

In mutex wait scheme 1 session repeatedly requests 1 ms sleeps:

```
kgxSharedExamine(...)
yield()
pollsys() timeout=1 ms repeated 25637 times
```

The `_mutex_wait_time` parameter controls the sleep timeout in *milliseconds*. This scheme differs from patch 6904068 by one additional *spin-and-yield* cycle at the beginning and smaller timeout.

Performance of this scheme is sensitive to `_mutex_wait_time` tuning. fig. 8 demonstrates that at moderate concurrency the short mutex sleeps performs better and results in bigger throughputs. However, such millisecond sleep will be rounded to centisecond on platforms like Solaris, Windows and latest HP-UX. This is because most OS can not sleep for very short times.

MVA estimation for mutex wait scheme 1 results in:

$$W_1 \approx \frac{k\rho}{1 - k\rho} (kT + T_r).$$

You see that additional spin at the beginning effectively reduces wait time T multiplying it by $k \ll 1$. This increases the performance.

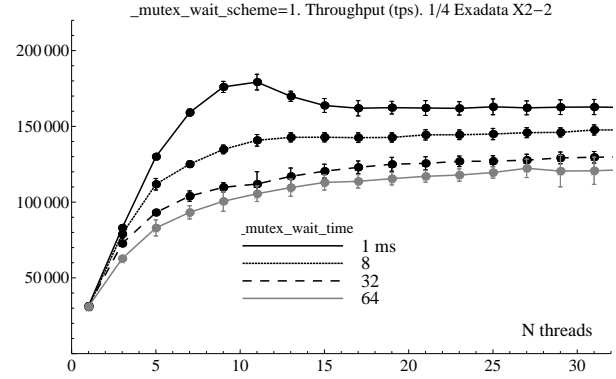


Fig. 8. Mutex wait scheme 1 throughput.

Default "Exponential Backoff" scheme 2

Oracle uses the scheme 2 by default. This scheme is named "Exponential backoff" in documentation. Unlike the previous versions, contemporary mutex wait do not consumes CPU. Surprisingly, DTrace shows that there is no exponential behavior by default. Session repeatedly sleeps with 1 cs duration:

```
yield() call repeated 2 times
semsys() timeout=10 ms repeated 4237 times
```

To reveal exponential backoff one need to increase the `_mutex_wait_time` parameter.

```
SQL> alter system set "_mutex_wait_time"=30;
...
yield() call repeated 2 times
semsys() timeout=10 ms repeated 2 times
semsys() timeout=30 ms repeated 2 times
semsys() timeout=80 ms
semsys() timeout=70 ms
semsys() timeout=160 ms
semsys() timeout=150 ms
semsys() timeout=300 ms repeated 159 times
```

This closely resembles the Oracle 8i latch acquisition algorithm [30, 29]. In scheme 2 the `_mutex_wait_time` controls maximum wait time in *centiseconds*. Due to exponentiality the mutex wait scheme 2 is insensitive to its value. Indeed, only sleep after the fifth unsuccessful spin is affected by this parameter [29].

Default mutex scheme 2 wait differs from patch 6904068 by two `yield()` syscalls at the beginning. These two *spin-and-yields* change the mutex wait performance drastically (fig. 9). They effectively multiply centisecond wait time T by k^2 :

$$W_2 \approx \frac{k\rho}{1 - k\rho} (k^2T + T_r).$$

Classic YIELDS. Mutex wait scheme 0

The `_mutex_wait_scheme` 0 consists mostly of repeating *spin-and-yield* cycles.

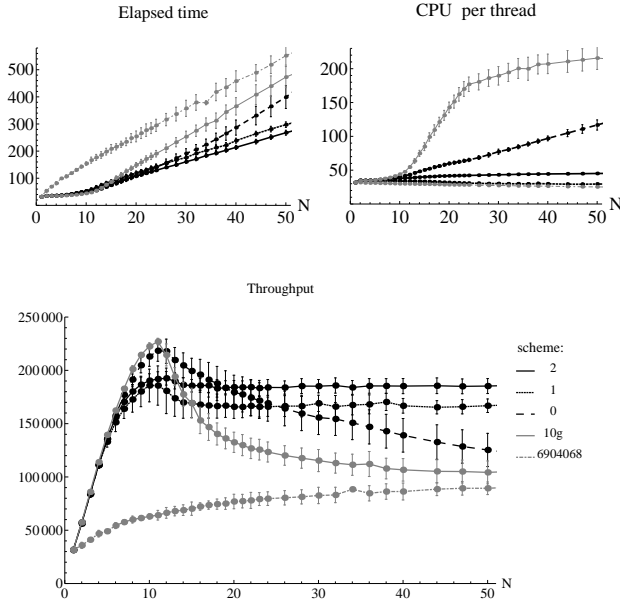


Fig. 9. Mutex wait schemes performance.

```
yield() call repeated 99 times
pollsys() timeout=1 ms
yield() call repeated 99 times
pollsys() timeout=1 ms
...
```

It differs from aggressive mutex waits used in previous Oracle versions by 1ms sleep after each 99 yields. This sleep significantly reduces CPU consumption and increases robustness. Unfortunately previous MVA style analysis is not applicable for this wait scheme.

The scheme 0 is very flexible [29]. The sleep duration and yield frequency are tunable by `_wait_yield_sleep_time_msecs` ... parameters. One can also specify different wait modes for standard and high priority processes. This allows almost any combination of yield and sleeps including 10g and patch 6904068 behaviors.

Comparison of mutex wait schemes

Fig. 9 compares performance of "Library Cache" mutex contention testcase on Exadata platform for all 3 wait schemes and the *patch 6904068* and *10g* mutex algorithms as well.

The figures demonstrate that:

Default scheme 2 is well balanced in all concurrency regions.

Wait scheme 1 should be used when the system is constrained by CPU.

Wait scheme 0 has the throughput close to 10g in medium concurrency region and may be recommended in case of plethora of free CPU.

10g mutex algorithm had the fastest performance in medium concurrency workloads. However, its throughput fell down when number of contending threads exceeds number of CPU cores. CPU consumption increased rapidly beyond this point. This

excessive CPU consumption starves processors and impacts other database workloads.

Patch 6904068 results in very low CPU consumption, but the largest elapsed time and the worst throughput.

IV. Mutex Contention

Mutex contention occurs when the mutex is requested by several sessions at the same time. Diagnosing mutex contention we always should remember Little's law

$$U = \lambda S$$

Therefore, the contention can be consequence of either:

Long mutex holding time S due to, for example, high SQL version count, bugs causing long mutex holding time or CPU starvation and preemption issues.

Or it may be due to high mutex exclusive Utilization. Mutexes may be overutilized by too high SQL and PL/SQL execution rate or bugs causing excessive requests.

Mutex statistics help to diagnose what actually happens.

Latest Oracle versions include many fixes for mutex related bugs and allow flexible wait schemes, adjustment of spin count and cloning of hot library cache objects. My blog [29] continuously discusses related enhancements.

Traditionally tuning of mutex performance problems was focused on changing the application and reducing the mutex demand. To achieve this one need to tune the SQL operators, change the physical schema, raise the bug with Oracle Support, etc... [16, 17, 18, 28].

However, such tuning may be too expensive and even require complete application rewrite. This article discusses one not widely used tuning possibility - changing of mutex spin count. This was commonly treated as an old style tuning, which should be avoided by any means. The public opinion is that increasing of spin count leads to waste of CPU. However, nowadays the CPU power is cheap. We may already have enough free resources. It makes sense to know when the spin count tuning may be beneficial.

Mutex spin count tuning

Long mutex holding time may cause the mutex contention. Default `_mutex_spin_count` =255 may be too small. Longer spinning may alleviate this. If the mutex holding time distribution has exponential tail:

$$\begin{aligned} Q(t) &\sim C \exp(-t/\tau) \\ k &\sim C \exp(-t/\tau) \\ \Gamma &\sim S_r - C\tau \exp(-t/\tau) \end{aligned}$$

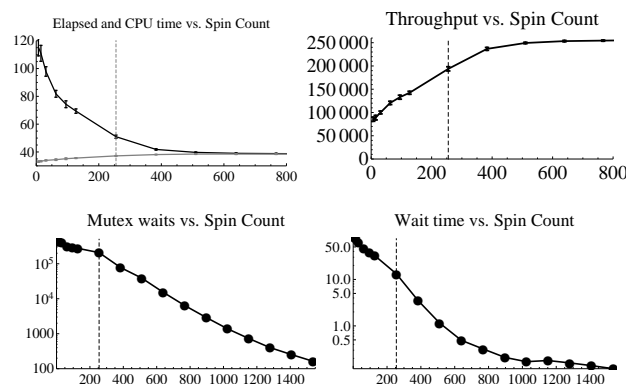


Fig. 10. Spin count tuning.

It is easy to see that if "sleep ratio" is small enough ($k \ll 1$) then

Doubling the spin count will square the "sleep ratio" and will only add part of order of k to spin CPU consumption.

In other words, if the spin is already efficient, it is worth to increase the spin count. Fig. 10 demonstrates effect of spin count adjustment for the "Library Cache" mutex contention test case.

The spin count tuning is very effective. Elapsed time fell rapidly while CPU increased smoothly. The number of mutex waits demonstrates almost linear behavior in logscale. This confirms the scaling rule.

Conclusions

This work investigated the possibilities to diagnose and tune mutexes, retrieval Oracle spinlocks. Using DTrace, it explored how the mutex works, its spin-waiting schemes, corresponding parameters and statistics. The mathematical model was developed to predict the effect of mutex tuning.

The results are important for performance tuning of highly loaded Oracle OLTP databases.

Acknowledgements

Thanks to Professor S.V. Klimenko for kindly inviting me to MEDIAS 2012 conference.

Thanks to RDTEX CEO I.G. Kunitsky for financial support. Thanks to RDTEX Technical Support Centre Director S.P. Misiura for years of encouragement and support of my investigations.

Thanks to my colleagues for discussions and all our customers for participating in the mutex troubleshooting.

References

- [1] Oracle®Database Concepts 11g Release 2 (11.2). 2010.
- [2] E. W. Dijkstra. 1965. Solution of a problem in concurrent programming control. *Commun. ACM* 8, 9 (September 1965), 569-. DOI=10.1145/365559.365617
- [3] J.H. Anderson, Yong-Jik Kim, "Shared-memory Mutual Exclusion: Major Research Trends Since 1986". 2003.
- [4] T. E. Anderson. 1990. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Trans. Parallel Distrib. Syst.* 1, 1 (January 1990), 6-16. DOI=10.1109/71.80120
- [5] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (February 1991), 21-65. DOI=10.1145/103727.103729
- [6] M. Herlihy and N. Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN:978-0123705914. "Chapter 07 Spin Locks and Contention."
- [7] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Proc. Conf. on Dist. Computing Systems*, 1982.
- [8] G.I. Falin, J.G.C. Templeton. 1997. *Retrial Queues*, Chapman and Hall, London. ISBN 978-0412785504.
- [9] J.R. Artalejo, A. Gomez-Corral. 2008. *Retrial Queueing Systems: A Computational Approach*, Springer, ISBN: 978-3-540-78724-2.
- [10] Beng-Hong Lim and Anant Agarwal. 1993. Waiting algorithms for synchronization in large-scale multiprocessors. *ACM Trans. Comput. Syst.* 11, 3 (August 1993), 253-294. DOI=10.1145/152864.152869
- [11] Anna R. Karlin, Mark S. Manasse, Lyle A. McGeoch, and Susan Owicki. 1990. Competitive randomized algorithms for non-uniform problems. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms* (SODA '90). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 301-309.
- [12] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, and A. Vainshtein. 1994. Optimal strategies for spinning and blocking. *J. Parallel Distrib. Comput.* 21, 2 (May 1994), 246-254. DOI=10.1006/jpdc.1994.1056
- [13] Ryan Johnson, Manos Athanassoulis, Radu Stoica, and Anastasia Ailamaki. 2009. A new look at the roles of spinning and blocking. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware* (DaMoN '09). ACM, New York, NY, USA, 21-26. DOI=10.1145/1565694.1565700
- [14] T. E. Anderson, D. D. Lazowska, and H. M. Levy. 1989. The performance implications of thread management alternatives for shared-memory multiprocessors. *SIGMETRICS Perform. Eval. Rev.* 17, 1 (April 1989), 49-60. DOI=10.1145/75372.75378
- [15] **My Oracle Support**, Oracle official electronic online support service. <http://support.oracle.com>. 2011.
- [16] Lewis J. 2011. *Oracle Core: Essential Internals for DBAs and Developers*. Apress ISBN: 978-1430239543
- [17] Adams S. 1999. *Oracle8i Internal Services for Waits, Latches, Locks, and Memory*. O'Reilly Media. ISBN: 978-1565925984

- [18] Millsap C., Holt J. 2003. *Optimizing Oracle performance*. O'Reilly & Associates, ISBN: 978-0596005276.
- [19] Richmond Shee, Kirtikumar Deshpande, K. Gopalakrishnan. 2004. *Oracle Wait Interface: A Practical Guide to Performance Diagnostics & Tuning*. McGraw-Hill Osborne Media. ISBN: 978-0072227291
- [20] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. 2004. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '04)*. USENIX Association, Berkeley, CA, USA, 2-2.
- [21] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. 1991. Empirical studies of competitive spinning for a shared-memory multiprocessor. *SIGOPS Oper. Syst. Rev.* 25, 5 (September 1991), 41-55. DOI=10.1145/121133.286599
- [22] L. Kleinrock, Queueing Systems, Theory, Volume I, ISBN 0471491101. Wiley-Interscience, 1975.
- [23] Cox D. 1970. *Renewal Theory*. London: Methuen & Co. pp. 142. ISBN 0-412-20570-X.
- [24] Olver F., et. al. 2010. *NIST Handbook of Mathematical Functions*, National Institute of Standards and Technology and Cambridge University Press , ISBN 978-0-521-19225-5, <http://dlmf.nist.gov/6.20>
- [25] Luke Y. 1969. *The Special Functions and their Approximations*. Vol. 2, Academic Press, New York. ISBN 978-0-124-59902-4
- [26] Reiser, M. and S. Lavenberg, "Mean Value Analysis of Closed Multichain Queueing Networks," *JACM* 27 (1980) pp. 313-322
- [27] Artalejo J.R., Resing J.A.C., "Mean Value Analysis of Single Server retrial Queues". *APJOR* 27(3): 335-345. 2010
- [28] Tanel Poder blog. *Core IT for Geeks and Pros* <http://blog.tanelpoder.com>
- [29] Andrey Nikolaev blog, *Latch, mutex and beyond*. <http://andreynikolaev.wordpress.com>
- [30] Nikolaev A.S. 2011. Exploring Oracle RDBMS latches using Solaris DTrace. In *Proc. of MEDIAS 2011 Conf.* ISBN 978-5-88835-032-4. <http://arxiv.org/abs/1111.0594v1>

*

About the author

Andrey Nikolaev is an expert at RDTEx First Line Oracle Support Center, Moscow. His contact email is Andrey.Nikolaev@rdtex.ru.