Andrey Nikolaev
RDTEX, Russia

Exploring Oracle RDBMS latches (spinlocks) using Solaris DTrace

MEDIAS - 2011
May 8-15

RDTEX
INFORMATION TECHNOLOGIES

Who am I

- Andrey.Nikolaev@rdtex.ru
- *http://andreynikolaev.wordpress.com*
- Graduated from MIPT in 1987
- 1987-1996 at COMPAS group, IHEP, Protvino
- Currently at RDTEX, Oracle First Line Support company
- Specialize in Oracle performance tuning
- Over 20 years of Oracle related experience as a research scientist, developer, DBA, performance consultant, trainer …

Introduction

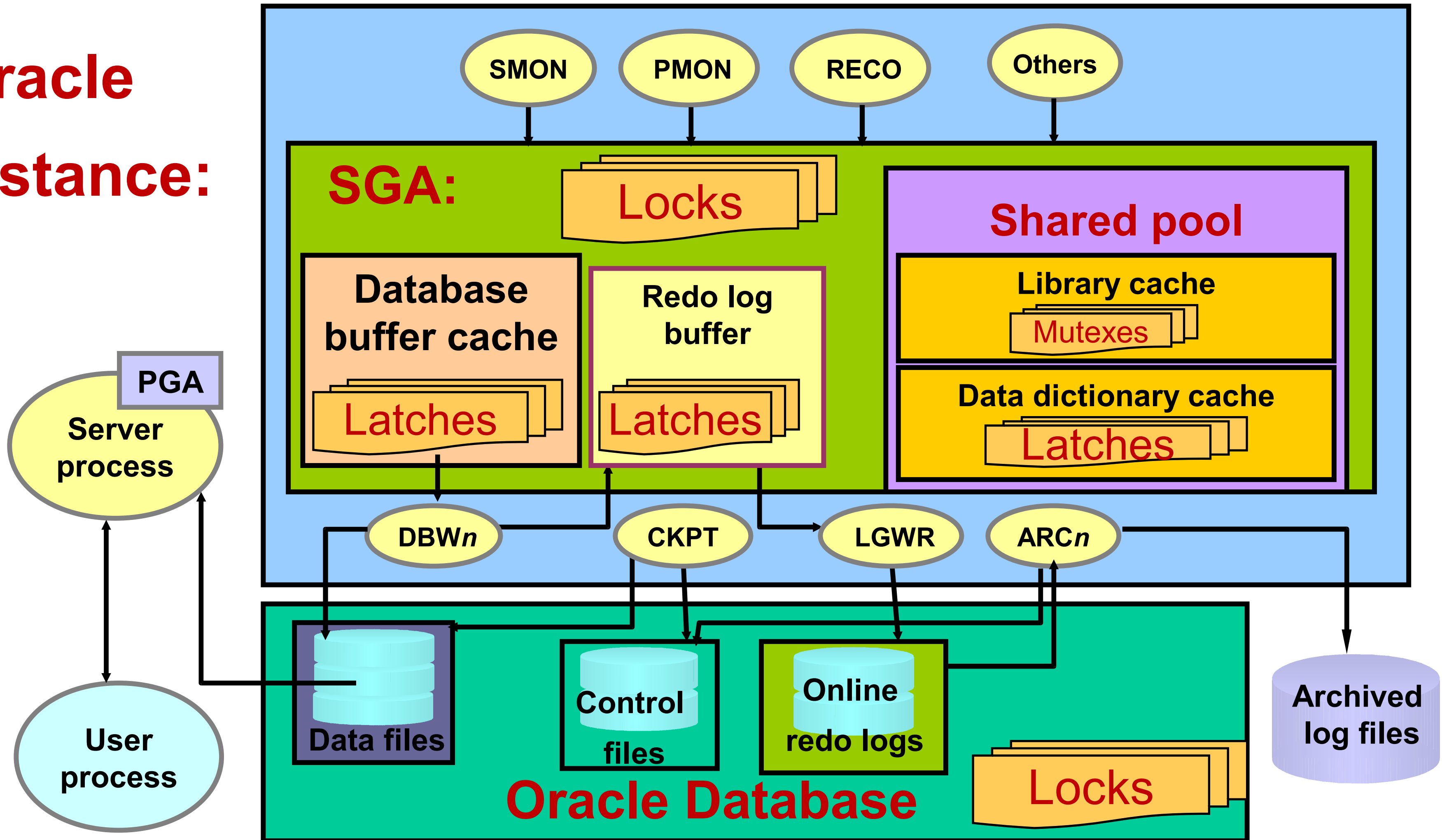
for non-Oracle

auditory

Oracle RDBMS
performance improvements timeline:

- 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, Distributing 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. 12c (2011): *?Cloud? Not yet released … to be continued*

Oracle Database Architecture: Overview



RDTEX INFORMATION TECHNOLOGIES

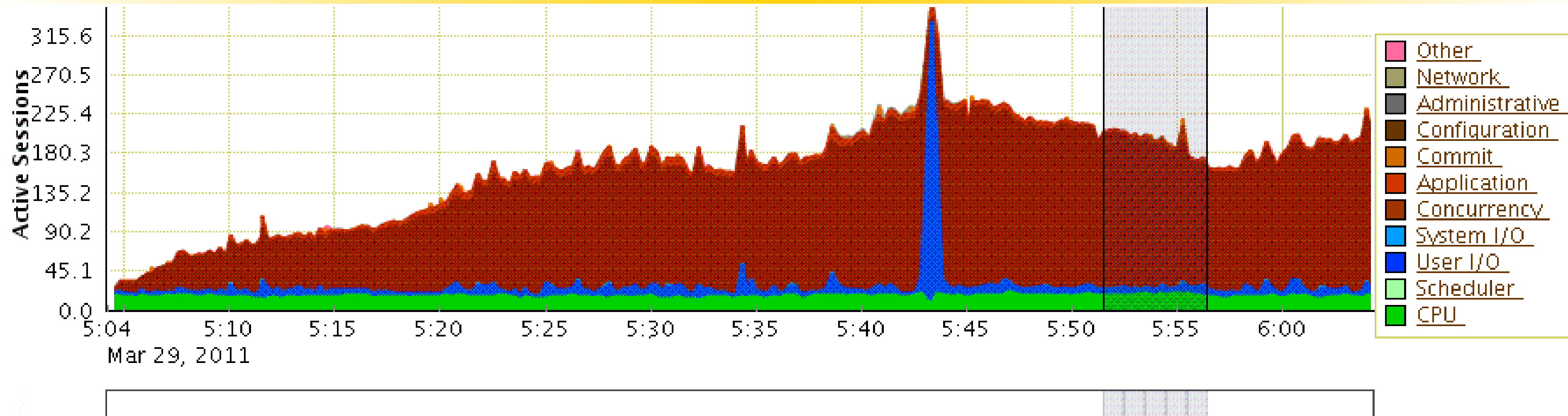
Why Oracle needs Performance Tuning?

- More then 100 books on Amazon. *Need for mainstream science support!*
- Complex and variable workloads. Every database is unique.
- Complex internals. 344 "Standard" / 2665 "Hidden" tunable parameters.
- Complicated physical database and schema design decisions.
- Concurrency and Scalability issues.
- Insufficient developers education.
- "Database Independence" issues.
- Self-tuning anomalies. SQL plan instabilities.
- OS and Hardware issues.
- More than 10 million bug reports on MyOracleSupport.

Oracle is well instrumented software:

- Oracle Statistics. "What sessions have done?". 628 statistics in 11.2.0.2
- Oracle Wait Interface. "How Oracle sessions have waited?". 1142 Wait events
- AWR/ASH/ADDM, Advisors, MyOracleSupport diagnostics and tuning tools, …
- Visualization challenge. *Oracle Enterprise Manager, Quest Spotlight, Embarcadero DB Optimizer, private tools, etc…*
- More than 2000 internal "dynamic performance" **X$** tables:
  - Needed for advanced diagnostics
  - Lack of documentation
  - Constantly changing.

Episode of latch contention:



Detail for Selected 5 Minute Interval

Start Time **Mar 29, 2011 5:51:29 PM MSD**

Top SQL

Schedule SQL Tuning Advisor | Create SQL Tuning Set

Select All | Select None

Select	Activity (%)	SQL ID	SQL Type
☐	35.54	406xnn0crnt9y	SELECT
☐	31.73	c7sd0s71s32hr	SELECT

Top Sessions

View Top Sessions

Activity (%)	Session ID	User Name
.51	1405	SIP_W
.51	863	SIP_W
.51	9	SIP_W

Oracle instance hangs due to heavy "**cache buffers chains**" latch contention

RDTEX INFORMATION TECHNOLOGIES

The presentation goals:

The goals of this work are:

- Explore one of Oracle serialization mechanisms: latches (spinlocks)
- Explore latch efficiency and possibilities of diagnostics and performance tuning.
- Explore how to interpret latch related performance counters.
- Explore latch spinning and waiting policies.
- Explore influence of Oracle parameters and adjustment of the number of spins for the latch before waiting

RDTEX
INFORMATION TECHNOLOGIES

Review of serialization mechanisms in Oracle

- *Latches are simple, low-level serialization mechanisms that coordinate multiuser access to shared data structures, objects, and files. …* ***Oracle® Database Concepts 11.2***

- Latch uses atomic hardware instructions for **Immediate Get**
- If **missed**, latch spins by polling location during **Spin Get**
- In spin get not succeed, latch sleeps for **wait get**.
- KGX Mutexes appeared in latest Oracle versions inside Library Cache only

_	Locks	Latches	Mutexes
Access	**Several Modes**	**Types and Modes**	***Operations***
Acquisition	**FIFO**	**SIRO (spin) + FIFO**	**SIRO**
SMP Atomicity	**No**	**Yes**	**Yes**
Timescale	**> Milliseconds**	**Microseconds**	**SubMicroseconds**
Life cycle	**Dynamic**	**Static**	**Dynamic**

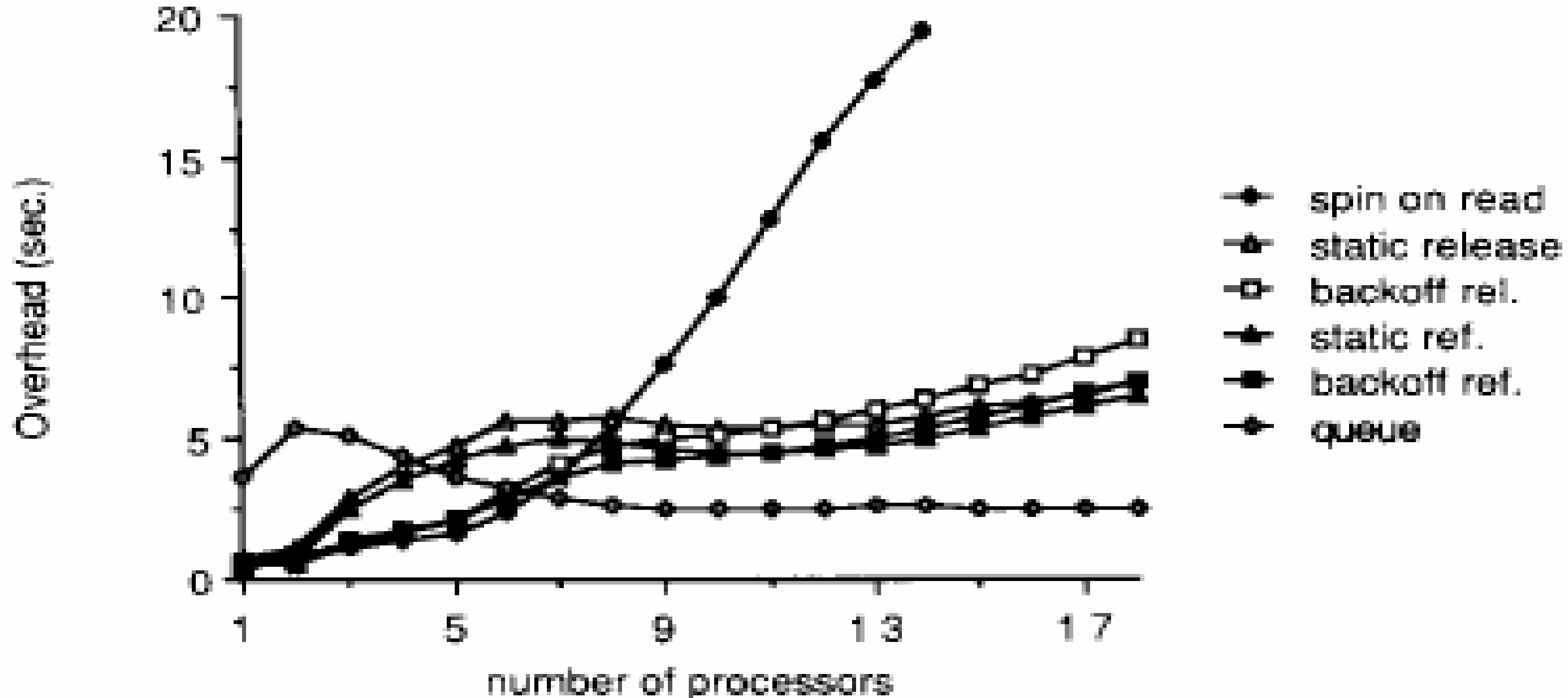
Classic spinlocks

- **Wiki**: "… spinlock … waits in a loop repeatedly checking until the lock becomes available …"
- Introduced by Edsger Dijkstra in **"Solution of a Problem in Concurrent Programming Control"** CACM. 1965
- Have been thoroughly investigated since that time. See "**The Art of Multiprocessor Programming"**, M. Herlihy and N. Shavit, Chapter 07 Spin Locks and Contention
- Many sophisticated spinlock realizations were proposed and evaluated (TS, TTS, Delay, MCS, Anderson,...) for high bus utilization ~100%
- Two general types:
  - **System spinlock**. Kernel OS threads cannot wait. Major metrics: atomic operations frequency. Shared bus utilization.
  - **User spinlock**. Oracle latch and mutex. Average lock holding time ~ 10 musec. It is more efficient to poll a lock rather than pre-empt the thread doing 1 msec context switch. Metrics: CPU and elapsed times.

Spinlock realizations

Spinlock:	Pseudocode:	Problems:
TS pre-11.2 mutex	**while(Test_and_Set(lock));**	Bus saturation by atomic operations
TTS Oracle latch	**while(lock||Test_and_Set(lock));**	Invalidation storms ("open door", "thundering herds").
Delay Mutex with patch 6904068	**Adjustable delay after noticing that lock was released**	Higher elapsed time under contention
Anderson, MCS, etc.	**Queues. Widely used in Java, Linux kernel … not in Oracle**	CPU and memory overhead, preemption issues

Anderson (1990) system spinlocks tests:



T.E. Anderson, “The Performance of Spin-Lock Alternatives for Shared-Memory Multiprocessors,” IEEE Trans. Parallel and Distributed Systems, Vol. 1, No. 1, Jan. 1990, pp. 6-16.

RDTEX
INFORMATION TECHNOLOGIES

DTrace. Solaris 10 Dynamic Tracing framework:

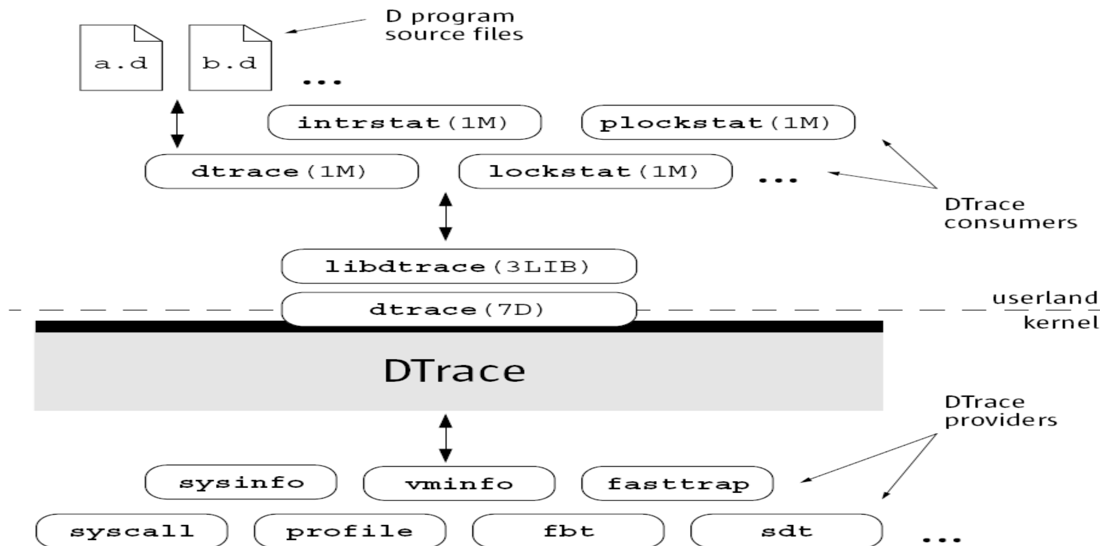
- Event-driven, kernel-based instrumentation allows to see all OS activity
- Dynamically interpreted C-like language to customize profiling
- No application changes needed to use DTrace
- Define the **probes** (triggers) to trap and write the handlers (**actions**).
- A lot of probes in Solaris kernel and ability to instrument every user instruction:

provider:module:function:name

pid1910:oracle:kslgetl:entry

- A **provider** is a methodology for instrumenting the system: **pid, fbt, syscall, sysinfo, vminfo …**
- **Action** is D routine to execute when a probe is hit
- **Predicates** define criteria for actions.

DTrace as a stroboscopic light:



DTrace allows us to investigate how Oracle latches perform in real time:

- Count the latch spins
- Trace how the latch waits
- Measure times and distributions
- Compute additional latch statistics

DTrace reveals latch interface routines:

Oracle calls the following functions to acquire the latch:

- **kslgetl(laddr, wait, why, where)** - get exclusive latch
- **kslgetsl (laddr,wait,why,where,mode)** - get shared latch
- …
- **kslfre(laddr)** - free the latch

Oracle give us possibility to do the same by **oradebug call**

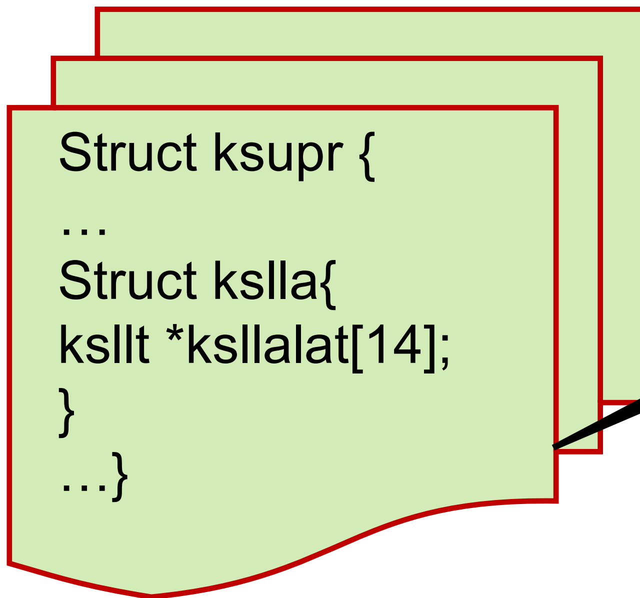
Function arguments meaning:

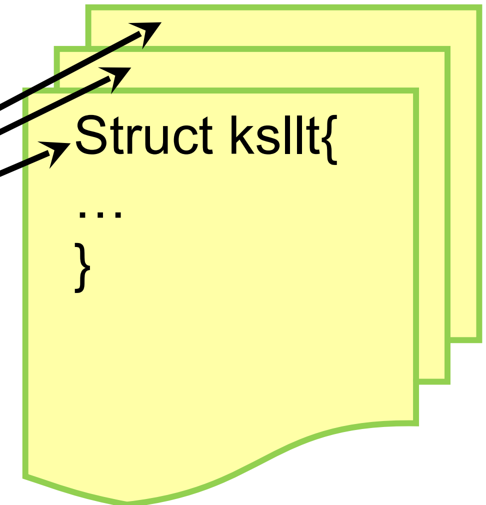
- **laddres** – address of latch in SGA
- **wait** – flag for no-wait or wait latch acquisition
- **where** – integer code for location from where the latch is acquired.
- **why** - integer context of why the latch is acquiring at this “where”.
- **mode** – requesting state for shared lathes. 8 – SHARED mode. 16 – EXCLUSIVE mode

Latch is holding by process, not session:

Process fixed array:
v$process -> x$ksupr

List of all latches:
v$latch ->x$ksllt





Each process has an array of references to the latches it is holding

Process latching info is the **kslla** structure embedded in the process state object

RDTEX INFORMATION TECHNOLOGIES

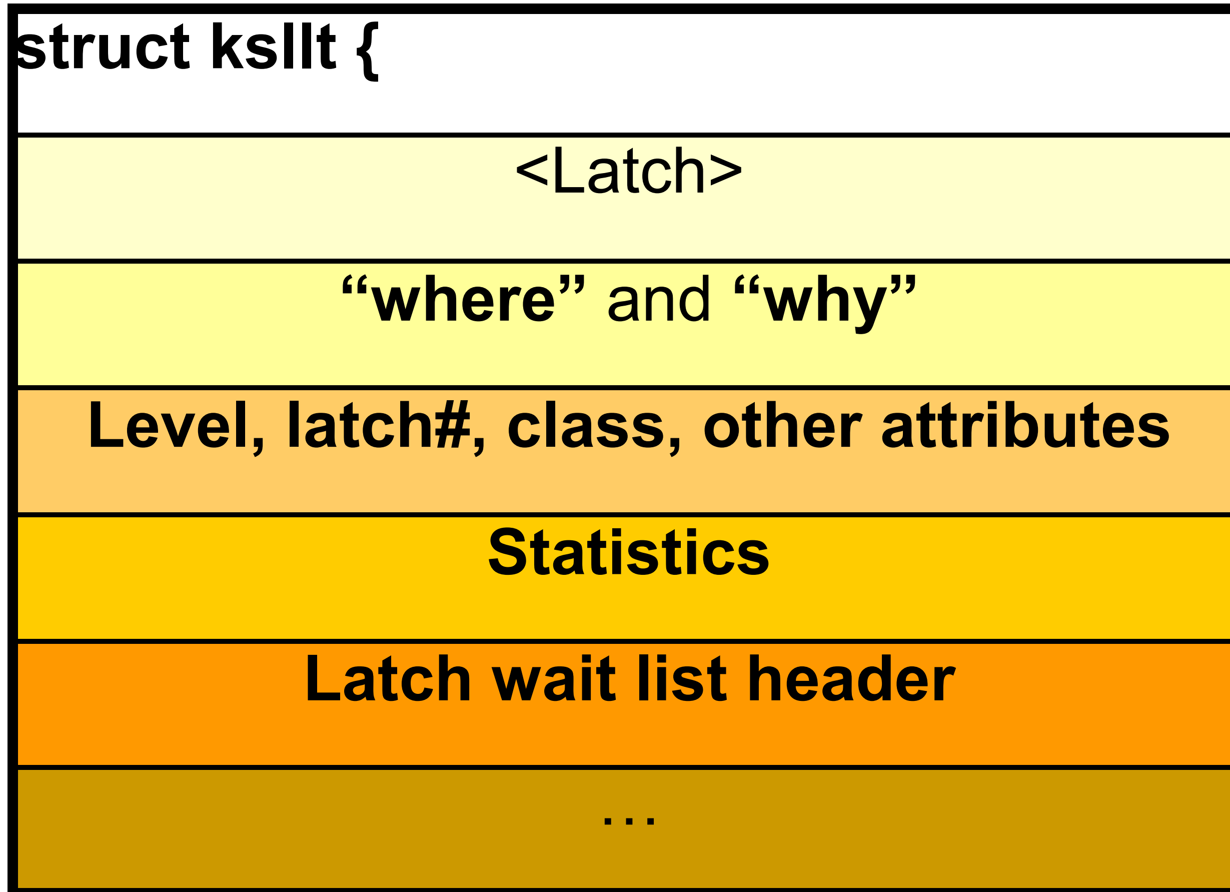
The latch get instrumentation:

X$KSUPR.KSLLA% fields instrument the latch get:

- **ksllalaq** – address of latch acquiring. Populated during immediate get (and spin before 11g)
- **ksllawat** - latch being waited for. This is **v$process.latchwait**
- **ksllawhy** – “**why**” for the latch being waited for
- **ksllawere** – “**where**” for the latch being waited for
- **ksllalow** – bit array of levels of currently holding latches
- **ksllaspn** - latch this process is spinning on. **v$process.latchspin**. Not populated since 8.1
- **ksllaps%** - inter-process post statistics

RDTEX
INFORMATION TECHNOLOGIES

The latch structure – kslit:



Latch size by version:

x$ksmfsv – list of all fixed SGA variables:

```
SELECT DISTINCT ksmfssiz
FROM x$ksmfsv
WHERE ksmfstyp = 'ksllt';
```

	*nix 32bit	*nix 64bit	Windows 32bit
7.3.4	92	-	120
8.0.6	104	-	104
8.1.7	104	144	104
9.0.1	?	200	160
9.2.0	196	240	200
10.1.0	**?**	**256**	**208**
10.2.0 - 11.2.0.2	100	160	104

Latch structure was bigger in 10.1 due to additional latch statistics

Oracle latch is not just a single memory location:

- ✓ Before 11g**.** Value of **first latch byte** (word for shared latches) was used to determine latch state:

0x00 – latch is free

0xFF – exclusive latch is busy. Was **0x01** in Oracle 7

0x01,0x02,… - shared latch holding by 1,2, … processes simultaneously

0x20000000 | pid - shared latch holding exclusively

- ✓ In 11g **first latch word** show the **pid** of the latch holder

0x00 – latch free

0x12 – Oracle process with pid 18 holds the exclusive latch

Latch attributes

Each latch have at least the following attributes in **kslldt** :

- ✓ **Name** Latch name as appeared in V$ views
- ✓ **SHR**. Is the latch Shared? Shared latch is “Read-Write” spinlock.
- ✓ **PAR**. Is the latch Solitary or Parent for the family of child latches?
- ✓ **G2C**. Can two child latches be simultaneously requested in wait mode
- ✓ **LNG**. Is wait posting used for this latch? Obsolete since Oracle 9.2.
- ✓ **UFS.** Is the latch Ultrafast? It will not increment miss statistics when STATISTICS_LEVEL=BASIC. 10.2 and above
- ✓ **Level.** 0-14. To prevent deadlocks latches can be requested in only in increasing level order
- ✓ **Class.** 0-7**.** Spin and wait class assigned to the latch. *9.2* and above.

Latches by Oracle version

Oracle version	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
10.2.0.2	385	114	55	-	4	47
10.2.0.3	388	117	58	-	4	48
10.2.0.4	394	117	59	-	4	50
11.1.0.6	496	145	67	-	6	81
11.1.0.7	502	145	67	-	6	83
11.2.0.1	535	149	70	-	6	86

RDTEX
INFORMATION TECHNOLOGIES

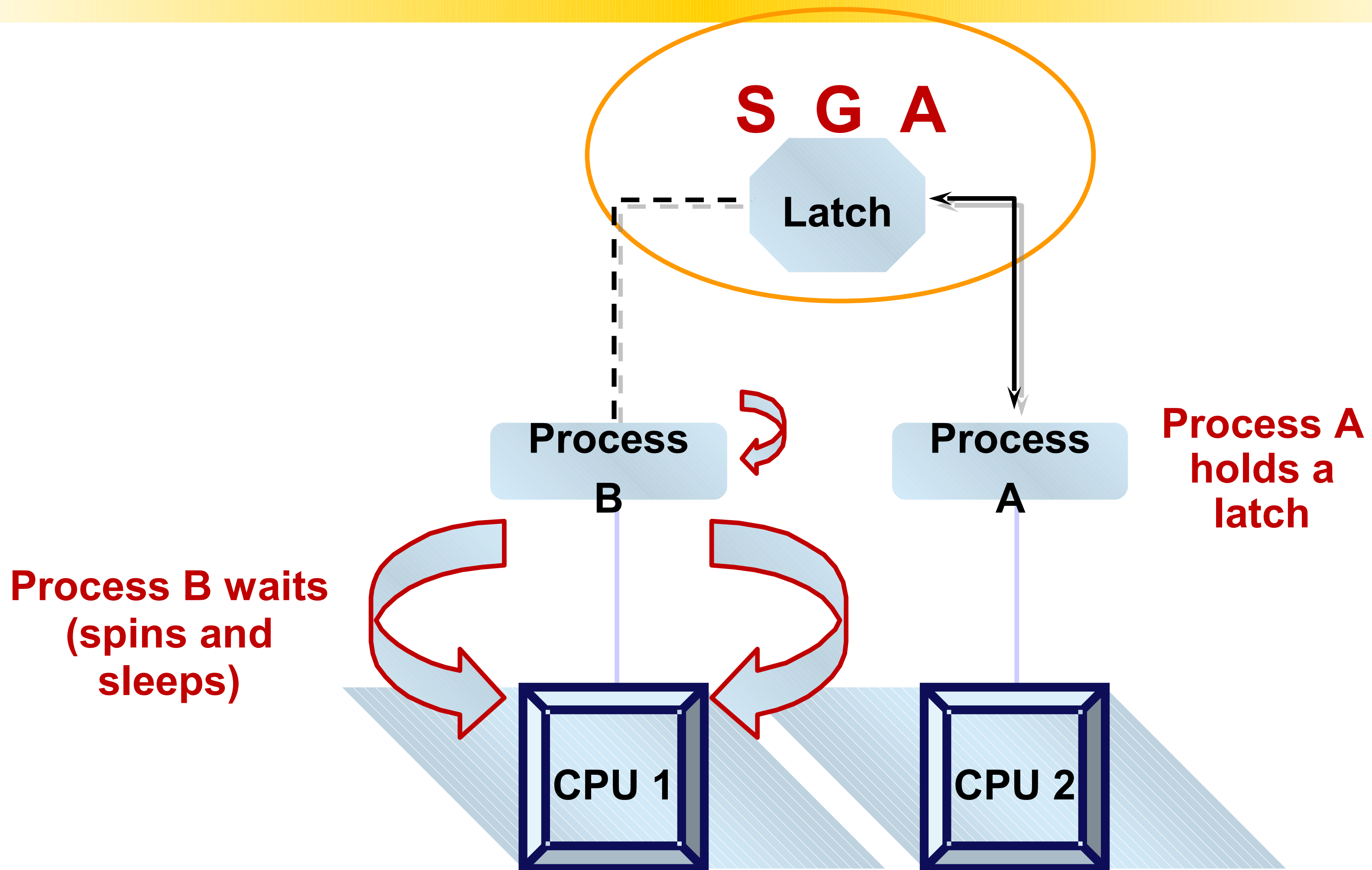
Latch trees

"**Rising level**" rule leads to "**trees**" of processes waiting for and holding the latches:

```
ospid: 28067    sid: 1677 pid: 61
 holding: 3800729f0     'shared pool' (156) level=7 child=1 whr=1602            kghupr1
      waiter: ospid: 129        sid: 72 pid: 45
       holding: a154b7120       'library cache' (157) level=5 child=17 whr=1664  kglupc: child
         waiter: ospid: 18255   sid: 65 pid: 930
         waiter: ospid: 6690    sid: 554 pid: 1654
         waiter: ospid: 4685    sid: 879 pid: 1034
         …
      waiter: ospid: 29749      sid: 180 pid: 155
       holding: a154b7db8       'library cache' (157) level=5 child=4 whr=1664   kglupc: child
         waiter: ospid: 13104   sid: 281 pid: 220
         waiter: ospid: 24089   sid: 565 pid: 636
         waiter: ospid: 25002   sid: 621 pid: 1481
         waiter: ospid: 16930   sid: 1046 pid: 783
```

Direct SGA access program output for 9.2.0.6 instance with too small shared pool.

Waiting for the latch

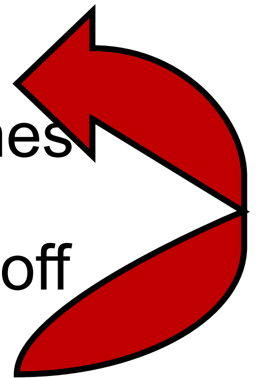


Latch Acquisition in Wait Mode

Version from contemporary 11.2 documentation. Was really used ten years ago in Oracle 7.3-8.1

Latch wait get (**kslgetl(laddress,1,…)**):

- One fast **Immediate get**, no spin
- **Spin get**: check the latch upto _SPIN_COUNT times
- **Sleep** on **"latch free"** event with exponential backoff
- Repeat



8i Latch get code flow using Dtrace

```
kslgetl(0x200058F8,1,2,3)          - KSL GET exclusive Latch# 29
 kslges(0x200058F8, ...)           - wait get of exclusive latch
   skgsltst(0x200058F8)     ...                  call repeated 2000 times = SPIN_COUNT
  pollsys(...,timeout=10 ms,...)     - Sleep 1
   skgsltst(0x200058F8)     ...                  call repeated 2000 times
  pollsys(...,timeout=10 ms,...)     - Sleep 2
   skgsltst(0x200058F8)     ...                  call repeated 2000 times
  pollsys(...,timeout=10 ms,...)     - Sleep 3
   skgsltst(0x200058F8)     ...                  call repeated 2000 times
  pollsys(...,timeout=30 ms,...)     - Sleep 4 …
```

- … **Event 10046 trace:**
- WAIT #0: nam='latch free' ela= 0 p1=536893688 p2=29 p3=0
- WAIT #0: nam='latch free' ela= 0 p1=536893688 p2=29 p3=1
- WAIT #0: nam='latch free' ela= 0 p1=536893688 p2=29 p3=2

RDTEX INFORMATION TECHNOLOGIES

Exponential backoff was inefficient

- 0.01-0.01-0.01-0.03-0.03-0.07-0.07-0.15-0.23-0.39-0.39-0.71-0.71-1.35-1.35-2.0-2.0-2.0-2.0...sec
- $timeout = 2^{[(N_{wait}+1)/2]} - 1$
- Typical latch holding time is 10 musec!
- Most waits were for nothing – latch already was free
- Latch utilization could not be more 70%
- Lot of unnecessary spins – provokes CPU thrashing

9.2-11g exclusive latch get flow using Dtrace

Semop – infinite wait until posted!

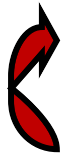
```
kslgetl(0x50006318, 1)
  -> sskgslgf(0x50006318)= 0          -immediate latch get
  -> kslges(0x50006318, ...)          -wait latch get
    -> skgslsgts(...,0x50006318, ...) -spin latch get
      ->sskgslspin(0x50006318)
        ...     - repeated 20000 cycles = 10*_SPIN_COUNT!
    -> kskthbwt(0x0)
    -> kslwlmod()                      - set up Wait List
    -> sskgslgf(0x50006318)= 0          -immediate latch get
    -> skgpwwait                       -sleep latch get
          semop(11, {17,-1,0}, 1)
```

Contemporary latch spins and waits

- Hidden latch wait revolution. In Oracle 9.2-11.2, all the latches in default class 0 **rely on wait posting**. Latch is sleeping without any timeout.
- If wakeup post is lost in OS, waiters will sleep infinitely.
- Latches assigned to non-default class **wait until timeout.**
- By default process spin **20000** cycles. Latch is **TTS** spinlock
- The _SPIN_COUNT parameter (by default 2000) is effectively static for exclusive latches.
- _LATCH_CLASS_0 initialization parameter determine exclusive latch wait and spin.

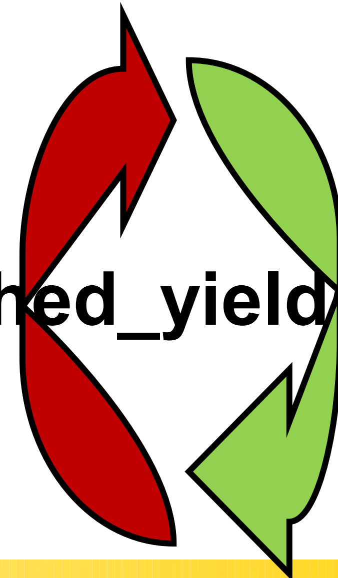
Nonstandard class latches

- Latch can be assigned to one of eight classes having different spin and wait policies. Standard class 0 latch use wait posting.
- **_LATCH_CLASS_X = "Spin Yield Waittime Sleep0 Sleep1 … Sleep7"**
- Nonstandard class latch loops upto "**Spin**" cycles, then yields CPU. This is repeated "**Yield**" times. Then the process sleeps for "**SleepX**" microseconds using **pollsys()** (not **semtimedop()**) system call.
  - If "**Yield**" !=0 repeat "**Yield**" times:



Loop up to "**Spins**" cycles

Yield CPU using **yield()** (or **sched_yield()**)



  - Sleep **for** "SleepX" usecs
  - Then spin again …

Shared latch acquisition

- Shared latch spin in Oracle 9.2-11g is governed by _SPIN_COUNT value and can be dynamically tuned
- **X** mode shared latch get spins by default up to 4000 cycles.
- **S** mode does not spin at all (or spins in unknown way)

	S mode get	X mode get
Held in S mode	Compatible	2*_SPIN_COUNT
Held in X mode	0	2*_SPIN_COUNT
Blocking mode	0	2*_SPIN_COUNT

Latch Release

- Free the latch – **kslfre(laddr)**
- Oracle process releases the latch **nonatomically**
- Then it sets up **memory barrier** – perform atomic operation on address individual to each process.
- This requires less bus invalidation and ensures propagation of latch release to other local caches.
- **Not fair policy** - spinners on the local CPU board have the preference.
- Then process posts first process in the list of waiters

The latch contention

Raw latch statistic counters

Statistics:	x$ksllt	Comments:
GETS	**kslltwgt**	"++" *after* **wait** mode latch get
MISSES	**kslltwff**	"++" *after* **wait** get if it was **missed**
SLEEPS	**kslltwsl**	"+**number_of_sleeps"** during get
SPIN_GETS	**ksllthst0**	"++" if get was **missed** but not **slept**
WAIT_TIME	**kslltwtt**	"+**wait_time"** *after* latch get
IMMEDIATE_GETS	**kslltngt**	"++" *after* **nowait** mode latch get. Is not protected by latch
IMMEDIATE_MISSES	**kslltnfa**	"++" if **nowait** mode get was missed
Wait queue length	L	Sampling of **x$ksupr.ksllawat**
N of spinning processes	N_s	Sampling of **x$ksupr.ksllalaq**

Differential (point in time) latch statistics

Latch requests arrival rate	$\lambda = \frac{\Delta gets}{\Delta time}$
Immediate gets efficiency	$\rho = \frac{\Delta misses}{\Delta gets}$
Latch sleeps ratio	$\kappa = \frac{\Delta sleeps}{\Delta misses}$
Latch wait time per second	$W = \frac{\Delta wait_time}{\Delta time}$
Latch spin efficiency	$\sigma = \frac{\Delta spin_gets}{\Delta misses}$

Should be calculated for each child latch. V$LATCH averaging distorts statistics

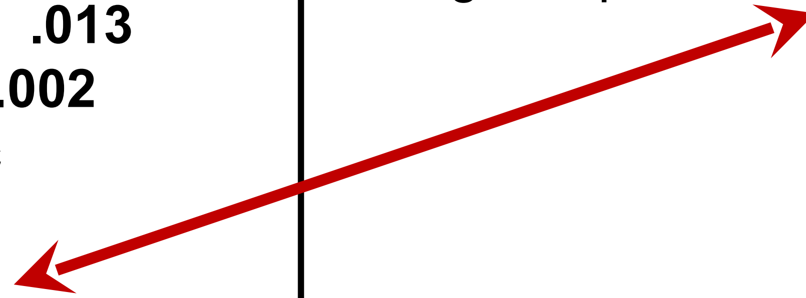
Derived latch statistics

Latch utilization: (PASTA)	$\rho \approx U = \frac{\Delta\, latch_holding_time}{\Delta\, time}$
Average holding time:	$S = \frac{\rho}{\lambda} = \frac{"Pct_Get_Miss" * "Snap_Time"}{100 * "Get_Requests"}$
Length of latch wait list:	$L = W$
Recurrent sleeps ratio:	$\frac{\sigma + \kappa - 1}{\kappa}$
Latch acquisition time:	$T_{aq} = \lambda^{-1}(N_s + W)$

RDTEX
INFORMATION TECHNOLOGIES

Latch statistics vs direct measurement

Latch statistics for:
0x380007358 "session allocation"
Requests rate: **lambda= 1350 Hz**
Miss /get: **rho= .022**
Sampled Utilization: **U= .013**
Slps /Miss: **kappa= .28**
Wait_time/sec: **W= .021**
Sampled queue length **Lw= .017**
Spin_gets/miss: **sigma= .72**
Sampled spinning procs:**Ns= .013**
Secondary sleeps ratio **= .002**
Avg holding time= **16.3** usec
sleeping time = **15.9** usec
acquisition time = **25.8** usec



Latch acquisition time distribution measured by DTrace:

```
--------- Distribution --------
    2048 |
    4096 |@@@@@@
    8192 |@@@@@@@@
   16384 |@@@@@@@@@@@@@@@@@@@@@@@
   32768 |@@@
   65536 |
      ns
```

Average acquisition time=**21** usec

RDTEX INFORMATION TECHNOLOGIES

Latch contention diagnostics in 9.2-11g

- Latch contention should be suspected if the latch wait events are observed in "**Top 5 Timed Events**" AWR section
- Look for the latch with highest W
- Symptoms of contention for the latch:
  - W > 0.1 sec/sec
  - Utilization ρ > 10%
  - Acquisition (or sleeping) time sufficiently greater then holding time
- **Latchprofx.sql** script invented by Tanel Poder greatly simplifies diagnostics.
- Script and **v$latch_misses** reveal "**where**" the contention arise
- Contention for a high-level latch frequently exacerbates contention for lower-level latches

RDTEX INFORMATION TECHNOLOGIES

Treating the latch contention:

- "**Right"** method: *tune the application and reduce the latch demand. Tune the SQL, bind variables, schema, etc…* Many brilliant books exist on this topic. Out of scope for this work.
- It may be too expensive and require **complete application rewrite**.
- Nowadays the CPU power is cheap. We may already have enough free CPU resources. **The spin count tuning may be beneficial.**
- Processes spin for exclusive latch spin upto 20000 cycles, for shared latch upto 4000 cycles and infinitely for mutex. Tuning may find more optimal values for **your application**.
- Oracle does not explicitly forbid spin count tuning. However, change of undocumented parameter should be discussed with Support.

Spin count adjustment

Shared latches:

- Spin count can be adjusted dynamically by _SPIN_COUNT parameter.
- Good starting point is the multiple of default **2000** value.
- Setting _SPIN_COUNT parameter in initialization file, should be accompanied by _LATCH_CLASS_0="20000". Otherwise spin for exclusive latches will be greatly affected by next instance restart.

Exclusive latches:

- Spin count adjustment by _LATCH_CLASS_0 parameter needs the instance restart.
- Good starting point is the multiple of default **20000** value.
- It may be preferable to increase the number of "yields" for class 0 latches.

Tuning spin count efficiently

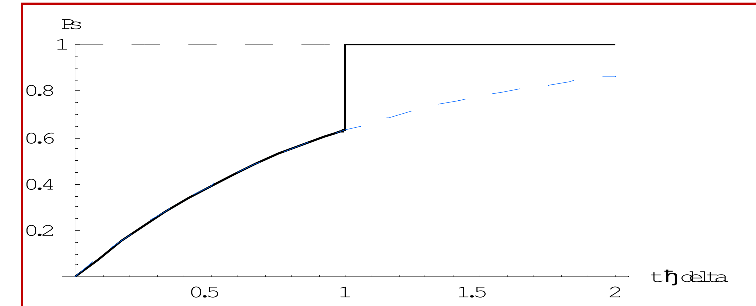
- First, the root cause of latch contention must be diagnosed.
- **Spin count tuning will only be effective if the latch holding time S is in its normal microseconds range**
- The number of spinning processes should remain far less then the number of CPUs. Analyze AWR and latch statistics before and after each change.
- It is a common myth that CPU time will raise infinitely while we increase spin count. Actually the process will spin up to "residual latch holding time"
- Elapsed time to acquire the latch will decrease while the latch "holding time" is less then OS "context switch time"

Latch spin CPU time

The spin probes latch holding time distribution. The spin time distribution is discontinuous at _SPIN_COUNT:

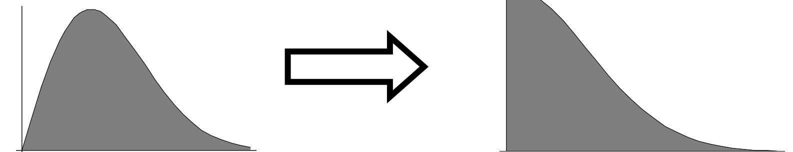
$$P_{sg}(t_s < t) = \begin{cases} P_l(\tau_k < t) & \text{when} \quad t < \Delta \\ 1 & \text{when} \quad t \geq \Delta \end{cases}$$

$$p_{sg} = p_l(t)H(\Delta - t) + (1 - P_l(\Delta))\delta(t - \Delta)$$



According to renewal theory distribution of time until the release is the transformed latch holding time distribution:

$$p_l(t) = \frac{1}{< t >}(1 - P(t)) = \frac{1}{< t >}Q(t)$$



Spin efficiency and average spin time are:

$$\begin{cases} \sigma = \frac{1}{\langle t \rangle} \int_0^\Delta Q(t)\mathrm{d}t \\ \Gamma_{sg} = \frac{1}{\langle t \rangle} \int_0^\Delta \mathrm{d}t \int_t^\infty Q(z)\,\mathrm{d}z \end{cases}$$

RDTEX INFORMATION TECHNOLOGIES

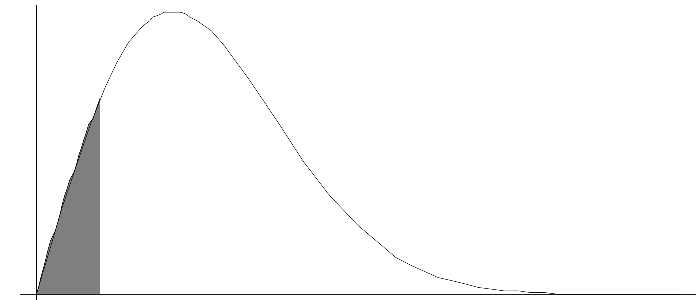
Spin count tuning when spin efficiency is low

To estimate effect of spin count tuning, we can use the approximate scaling rules depending on the value of:

σ = "spin efficiency"="Spin gets/Miss"

If the spin is inefficient $\sigma << 1$ then spin probes the latch holding time distribution around the origin:

$$\begin{cases} \sigma = \frac{\Delta}{\langle t \rangle} - \frac{1}{\langle t \rangle} \int_0^{\Delta} (\Delta - t) p(t)\, dt \\ \Gamma_{sg} = \Delta - \frac{\Delta^2}{2\langle t \rangle} + \frac{1}{2\langle t \rangle} \int_0^{\Delta} (\Delta - t)^2 p(t)\, dt \end{cases}$$



If processes do not release latch immediately:

Therefore:

$$\begin{cases} \sigma = \frac{\Delta}{\langle t \rangle} + O(\Delta^3) \\ \Gamma_{sg} = \Delta - \frac{\Delta^2}{2\langle t \rangle} + O(\Delta^4) \end{cases}$$

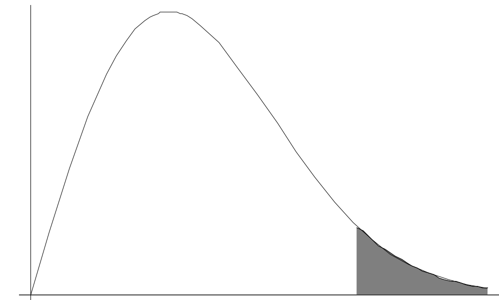
In this region doubling the spin count will double "spin efficiency" and also double the CPU consumption

RDTEX INFORMATION TECHNOLOGIES

Spin count tuning when efficiency is high

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

$$\begin{cases} \sigma = 1 - \frac{1}{\langle t \rangle} \int_{\Delta}^{\infty} (t - \Delta) p(t)\, dt \\ \Gamma_{sg} = \frac{\langle t^2 \rangle}{2\langle t \rangle} - \frac{1}{2\langle t \rangle} \int_{\Delta}^{\infty} (t - \Delta)^2 p(t)\, dt \end{cases}$$



Oracle normally operates in this region of small latch sleeps ratio $\kappa = 1 - \sigma < 0.1$

Here spin count is greater than number of instructions protected by latch $\Delta \gg \langle t \rangle$

The spin time is bounded by the "residual latch holding time" and spin count:

$$\Gamma_{sg} < \min\left(\frac{\langle t^2 \rangle}{2\langle t \rangle} , \Delta \right)$$

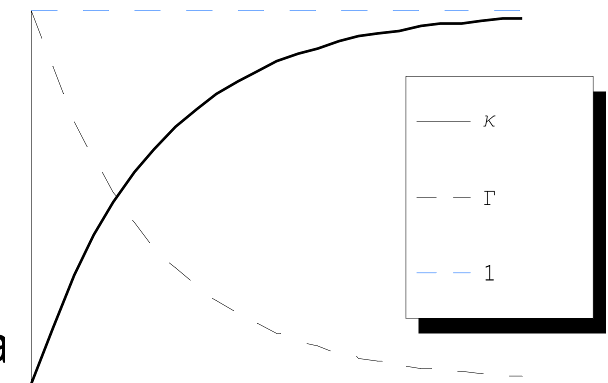
Sleep prevents latch from waste CPU for spinning for heavy tail of holding time distribution

Exponential tail spin scaling

- Experiments showed that normally latch holding time distribution has exponential tail: $Q(t) \sim C \exp(-t/\tau)$

$$\kappa = 1 - \sigma \sim C \exp(-t/\tau)$$

$$\Gamma_{sg} \sim \frac{\langle t^2 \rangle}{2\langle t \rangle} - C\tau \exp(-t/\tau)$$



- Compare this to Guy Harrison experimental data
- If "sleep ratio" is small $\kappa = 1 - \sigma << 0.1$ then:

Doubling the spin count will square the "sleep ratio" coefficient. This will only add part of order κ to spin CPU consumption

Oracle DBA paraphrase: If "sleep ratio" for exclusive latch is 10% than increase of spin count to **40000** may results in **10** times decrease of "latch free" wait events, and only **10%** increase of CPU consumption.

If the spin is already efficient, it is worth to increase the spin count.

RDTEX INFORMATION TECHNOLOGIES

Long distribution tails: CPU thrashing

- Latch contention can cause CPU starvation. Processes contending for a latch, also contend for CPU.
- Once CPU starves, OS runqueue length raise and **loadaverage** exceeds the number of CPUs. Some OS may shrink the time quantum. Latch holders will not receive enough time to release the latch.
- Due to priority decay, latch acquirers may preempt latch holders. This leads to **priority inversion.** The throughput falls.
- Transition to this stable state is more likely if workload of your system approaches ~100% CPU
- Due to preemption, **latch holding time S will raise** to the CPU scheduling scale.
- To prevent CPU thrashing use fixed priority OS scheduling classes.

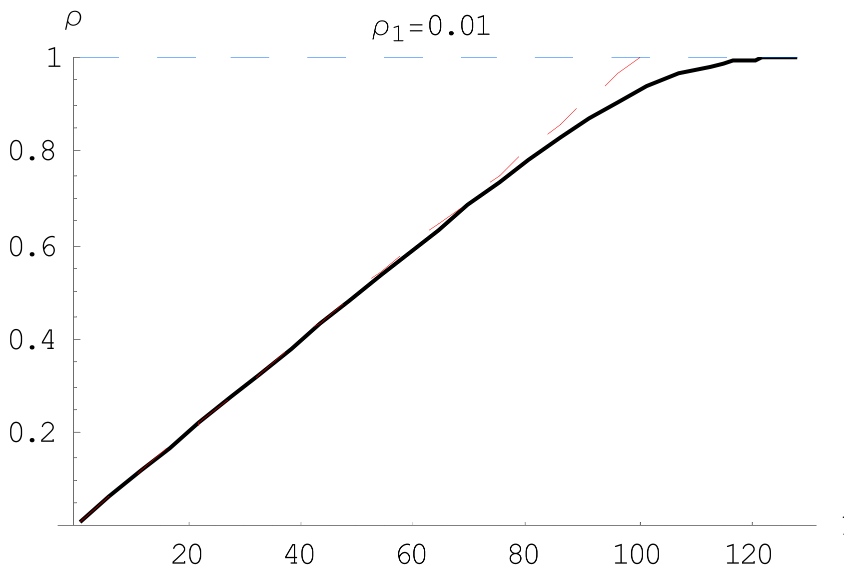
Latch SMP scalability

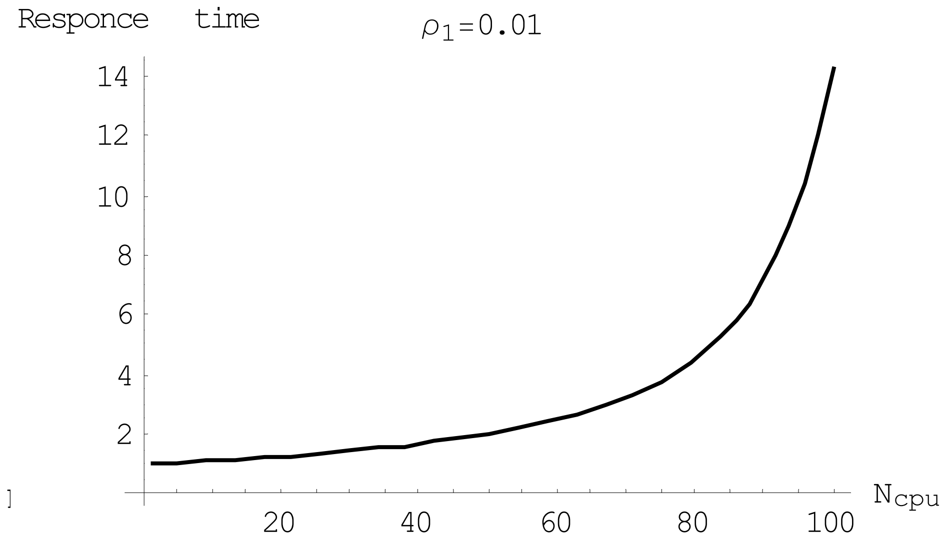
- If latch utilization is ρ_1 in single CPU environment.
- Then in N CPU server latch utilization will be $\rho_N \approx N\rho_1$. This can be problematic:
  - If single CPU system held latches only for 1% of time
  - 48 CPU server with the same per-CPU load will hold latches for 50%
  - 128 CPU Cores server will suffer huge latch (and mutex) contention
- This is also known as **"Software lockout"**. It may substantially affect contemporary multi-core servers.
- NUMA should overcome this intrinsic spinlock scalability restriction

Spinlock SMP scalability estimations

$$\rho_N = 1 - \left[\sum_{k=0}^{N} \left(\frac{\rho_1}{1-\rho_1} \right)^k \frac{N!}{(N-k)!} \right]^{-1}$$

$$W \sim \frac{\rho_N}{1-\rho_N} \bar{S}$$





B. Sinharoy, et al. , Improving Software MP Efficiency for Shared Memory Systems.
Proc. of the 29th Annual Hawaii International Conference on System Sciences – 1996

Q/A?

- Questions?
- Comments?

Acknowledgements

- Thanks to Professor S.V. Klimenko for kindly inviting me to MEDIAS 2011 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

Thank you!

Andrey Nikolaev

http://andreynikolaev.wordpress.com

Andrey.Nikolaev@rdtex.ru

RDTEX, Moscow, Russia

www.rdtex.ru