Troubleshooting Funny Issues

Real Life Case Studies

SOUG Day 02/2024

2024-09-20

Christoph Lutz
Thomas Mayer

Why this Session?

- Troubleshooting is a large part of our job and both, an **art and science** at the same time
- Complex problems can only be solved by following a **systematic approach**, which means we must understand them!
- Oracle offers a wealth of **diagnostics**, most of them not publically documented though
- Learn about new perspectives , tools and ideas to **become more effective at systematic troubleshooting**

A Word of Caution

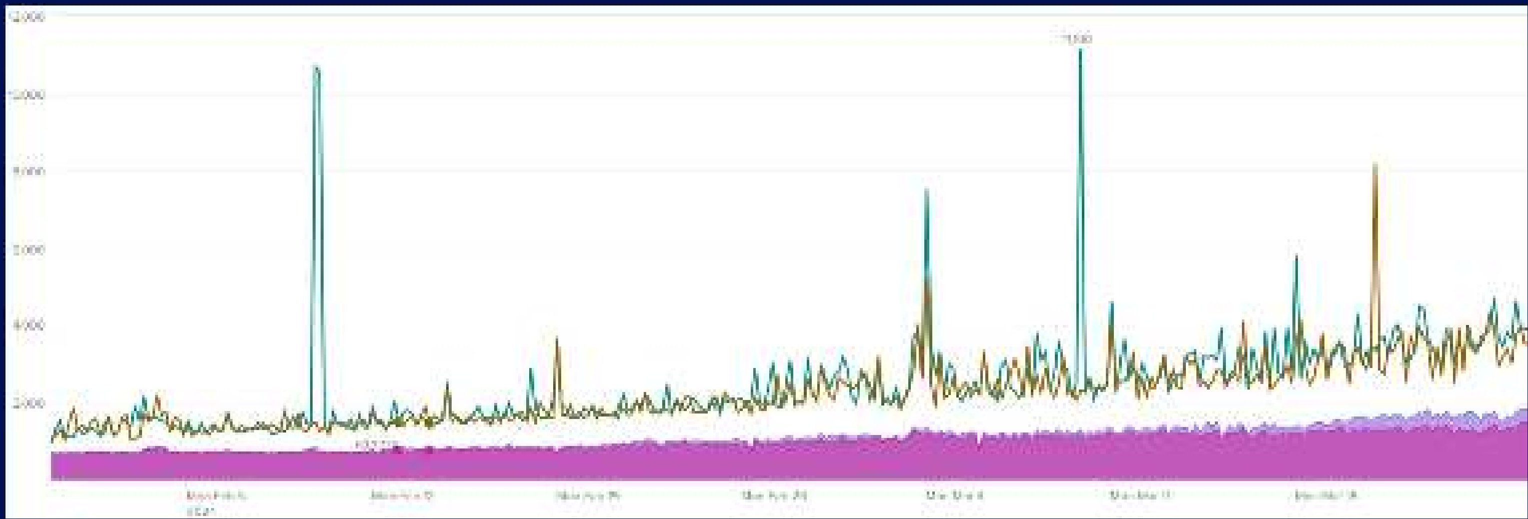
This is a low-level technical presentation about internal and undocumented behavior

Beware that:

- Things can change across different versions and patch levels
- My observations, findings and interpretations may be inaccurate or wrong
- Some of the techniques shown in this presentation are **dangerous** – use them at your own risk!

Case 1:
Shared Pool Latch Contention

Starting Situation – Problem Description



Connection test elapsed times increase over time.

Simple "**SELECT 1 FROM DUAL**" statement.

Starting Situation – Shared Pool Latch Contention (AWR)

Top 10 Foreground Events by Total Wait Time

Event	Waits	Total Wait Time (sec)	Avg Wait	% DB time	Wait Class
latch: shared pool	21,564	8050	373.31ms	54.8	Concurrency
DB CPU		3898.7		26.5	
cell single block physical read: flash cache	2,696,058	1015.2	376.56us	6.9	User I/O
enq: TX - index contention	173	976.2	5643.01ms	6.6	Concurrency
latch free	3,746	592.4	158.13ms	4.0	Other
enq: PS - contention	18,609	165.9	8.92ms	1.1	Other
cell multiblock physical read	38,001	54.2	1.43ms	.4	User I/O
cell smart table scan	44,700	51.2	1.15ms	.3	User I/O
gc cr multi block mixed	41,659	33.4	801.64us	.2	Cluster
cell list of blocks physical read	43,656	32.8	751.80us	.2	User I/O

Massive **shared pool latch** contention (54 % of total db time).

This is clearly not healthy!

Very long avg wait time (373 ms).

Is this related to the connection test slow down and how?

Shared Pool Latch Contention – Latch Miss Sources (V$LATCH_MISSES)

Latch Miss Sources

- only latches with sleeps are shown
- ordered by name, sleeps desc

Latch Name	Where	NoWait Misses	Sleeps	Waiter Sleeps
unknown latch	kghalo	0	20,353	21,733
unknown latch	kghfre	0	1,973	1,793
unknown latch	kghfnd: req scan	0	546	0
unknown latch	kghfnd: get next extent	0	325	0
unknown latch	kghfnd: min scan	0	299	0
unknown latch	kghfree_extents: scan	0	92	174
unknown latch	kghfru	0	86	1
unknown latch	kghupr1	0	78	184
unknown latch	kghfnd: resv scan	0	69	0
unknown latch	kghfrunp: no latch	0	52	0
unknown latch	kghalp	0	33	36
unknown latch	ksqcmi: if lk mode requested	0	18	6
unknown latch	ksqcmi: if lk mode not requested	0	14	7
unknown latch	ksqgtl3	0	14	30
unknown latch	kghfnd: min to max scan	0	11	0
unknown latch	ksltbl	0	8	1
unknown latch	kghasp	0	5	1
unknown latch	ksqrcl	0	5	8
unknown latch	ksqcnl	0	4	4
unknown latch	kgh_heap_sizes	0	2	0

The latch gets were caused by memory allocation and deallocation!

unknown latch
In some 19c RUs, the **shared pool latch** is wrongly displayed as "unknown latch" in the AWR Latch Miss Sources section.

Where
Code location where the latch is held (not request location)

kghalo Kernel Generic Heap Manager Allocate
=> Allocate a chunk of memory in the shared pool.

kghfre Kernel Generic Heap Manager Free
=> Free a chunk of memory in the shared pool.

Sleeps
Number of times that a process slept while the latch was **held** from this location (blocker information).

Waiter Sleeps
Number of times that a process slept while **requesting** the latch from this location (blockee information).

This AWR section only exposes counters, but no details on :

- latch hold time
- hot code paths resulting in a latch get

Shared Pool Latch Contention – Systematic Analysis with latchprofx

```
SQL> @latchprofx.sql sid,name,hmode,func % "shared pool" 100000


   SID NAME         HMODE        FUNC                 Held       Gets   Held %     Held ms    Avg hold ms
------ ------------ ------------ ------------------ ---------- ------- ------- ---------- -------------
  1043 shared pool  exclusive    kghfnd: req scan        69579      36   69.58  24359.608       676.656
  1043 shared pool  exclusive    kghalo                     47      47     .05     16.455          .350
  1043 shared pool  exclusive    kghfre                     17      17     .02      5.952          .350
    88 shared pool  exclusive    kghalo                     12      12     .01      4.201          .350
  1702 shared pool  exclusive    kghalo                     11      11     .01      3.851          .350
  1727 shared pool  exclusive    kghfre                     11      11     .01      3.851          .350
  1115 shared pool  exclusive    kghalo                     10      10     .01      3.501          .350
  …
```

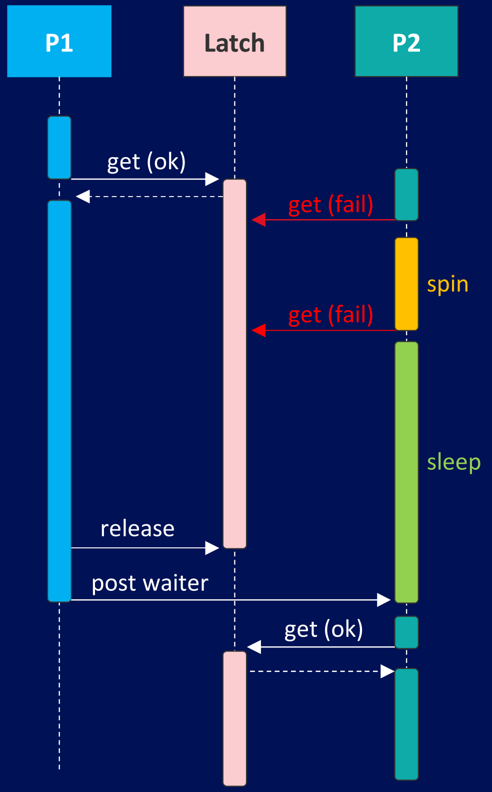
Session 1043 held the shared pool latch **69.5 %** of the time with an avg hold time of **~0.7 sec !**

kghfnd = Kernel Generic Heap manager FiND => find a free chunk of memory in the shared pool

Session 1043 was exclusively holding the shared pool latch while searching for free memory in the shared pool!

latchprofX.sql Script Source: Tanel Poder, TPT Github Repository, latchprofx.sql

What is a Latch?



Latches are Oracle's implementation of "adaptive spin-locks".

Historically, latches used an exponential back-off wait scheme.
This no longer applies to modern versions of Oracle!

At the instruction level, latches use an atomic cmpxchg instruction (on x86-64).

Oracle Latches – C Function Signatures

Exclusive Latch Acquisition

```
kslgetl(laddr, wait, why, where)
```

Shared Latch Acquisition

```
ksl_get_shared_latch(laddr, wait, why, where, mode, new_value)
```

Latch Release

```
kslfre(laddr)
```

Function Parameters

laddr: Address of latch in SGA

wait: flag for no-wait (0) or wait (1) mode

where: code location where latch is acquired (maps to **x$kslw.indx**)

why: Context and reason why latch is acquired at "where" (**x$kslw.ksllwlbl**)

mode: Requested state for shared latches (8=SHARED, 16=EXCLUSIVE)

new_value: value to determine latch state
0x1, 0x2, etc. – shared latch held by 1, 2, etc. processes
0x20000000 | pid – shared latch held exclusively

Function and parameter names source: Andrey Nikolaev, Latch internals, RUOUG Seminar, 2012-12-06

Shared Pool Latch Contention – Systematic Analysis with bpftrace (1/2)

```
select lower(addr) from v$latch
where name = 'shared pool'
union
select lower(addr) from v$latch_children
where name = 'shared pool'
/


ADDR
----------------
0000000060079380
00000000604746d8
0000000060474778
0000000060474818
00000000604748b8
0000000060474958
00000000604749f8
0000000060474a98
```

```
#define KSPSSIDST 0x60009628

#define LADDR0    0x60079380
#define LADDR1    0x604746d8
#define LADDR2    0x60474778
#define LADDR3    0x60474818
#define LADDR4    0x604748b8
#define LADDR5    0x60474958
#define LADDR6    0x604749f8
#define LADDR7    0x60474a98


uprobe:$ORACLE_HOME/bin/oracle:kslgetl
/ str(uptr(KSPSSIDST)) == str($1) /
{
    if (arg0 == LADDR0 ||
        arg0 == LADDR1 ||
        arg0 == LADDR2 ||
        arg0 == LADDR3 ||
        arg0 == LADDR4 ||
        arg0 == LADDR5 ||
        arg0 == LADDR6 ||
        arg0 == LADDR7 )
    {
       @[ustack()] = count();
    }
}
```

Collect and count all code paths (stack traces) that acquire one of the shared pool latches (for the given instance in $1).

Shared Pool Latch Contention – Systematic Analysis with bpftrace (2/2)

Example Script Output

```
 …
 @[
     kslgetl+0
     kghalo+5925
     ksp_param_handle_alloc+932
     kspcrec+228
     ksucre+822
     kxfpProcessJoin+1236
     kxfpProcessMsg+695
     kxfpqidqr+1524
     kxfprdp_int+1677
     opirip+619
     opidrv+581
     sou2o+165
     opimai_real+173
     ssthrdmain+417
     main+256
]: 2001
 …
```

The call stack sampling shows what code path called into kslgetl and how many times in total the code path got executed (by all Oracle processes of a particular instance).

Function Names & Prefixes

kslgetl – Kernel Service Layer Get Latch (exclusive latch get)
kghalo – Kernel Generic Heap Manager Allocate
ksp – Kernel Service Parameter
ksucre – Kernel Service User Create User Session
kxfp – Kernel eXecution Parallel Query Process
opi – Oracle Programm Interface

Call stacks can answer why the shared pool latch was requested.

But how can we efficiently analyze and aggregate thousands of different call stacks?

=> Flame Graphs!

Shared Pool Latch Contention – Flame Graphs Visualization

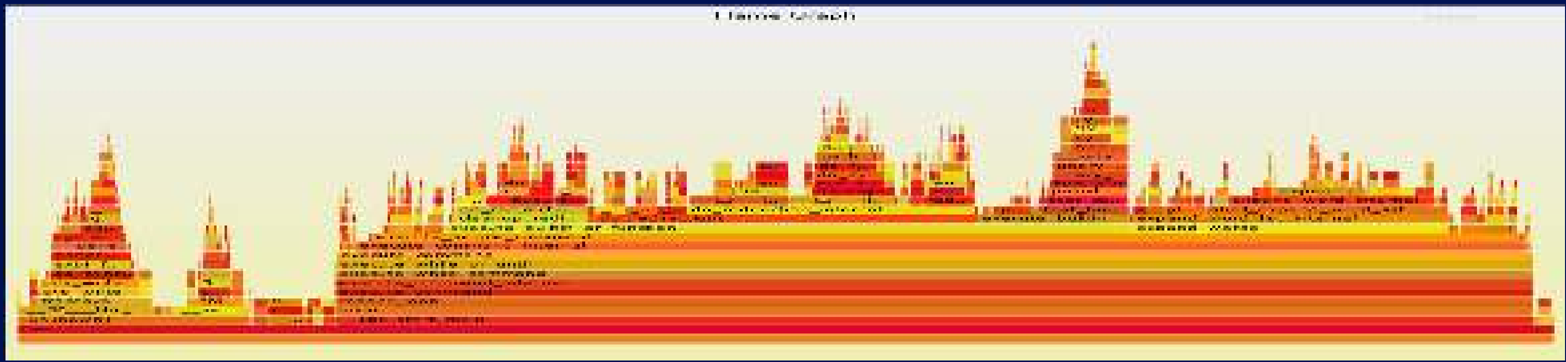
Idea

Visualize stack traces to identify frequent and "hot" code paths.

Interpretation

- **x-axis**: stack profile population. **This is not the passage of time!**
- **y-axis**: stack depth

The wider a frame, the more often it was present in the stacks. **Look for plateaus.**



Source: Brendan Gregg, Flame Graphs, 2020-10-31

Shared Pool Latch Contention – Flame Graphs Creation

bpftrace symbol lookups are costly and slow, therefore it is highly recommended to cache symbol lookups!

1. **Collect stack traces**

```
$ BPFTRACE_CACHE_USER_SYMBOLS=1 ./kslgetl.bt MY_ORACLE_SID > stacks.txt
```

2. **Collapse bpftrace call stacks**

```
$ ./stackcollapse-bpftrace stacks.txt > stacks-folded.txt
```

3. **Generate Flame Graph**

```
$ ./flamegraph.pl stacks-folded.txt > stacks.svg
```

Shared Pool Latch Contention – Flame Graphs

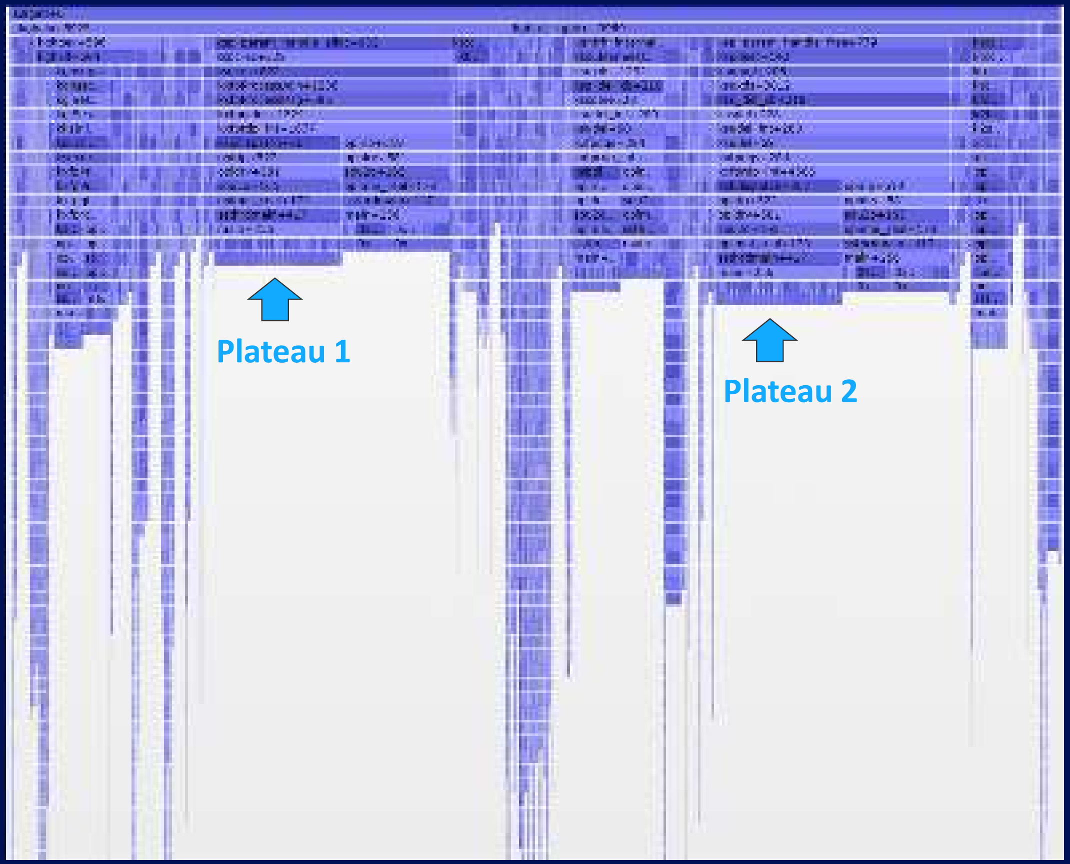
"Hairy graph" - no plateaus are standing out.

Activity can hardly be attributed to particular code paths.

This pattern typically occurs with lock contention.

We can also reverse the merge order of flame graphs (merge from leaf to root instead of root to leaf)

Shared Pool Latch Contention – Reverse Flame Graph



The **reverse graph** on the left now shows two plateaus standing out:

Plateau 1

```
kslgetl+0
kghalo+5925
ksp_param_handle_alloc+932
kspcrec+228
ksucre+822
kxfpProcessJoin+1236
kfxpProcessMsg+695
kxfpqidqr+1524
kxfprdp_int+1677
ksbdispatch+367
opirip+522
opidrv+581
sou2o+165
opimai_real+173
ssthrdmain+417
main+256
```

Shared pool memory allocation due to **spawning PX sessions.**

Plateau 2

```
kslgetl+0
kghfre+3989
ksp_param_handle_free+779
kspdesc+142
ksmugf+208
ksuxds+3812
kss_del_cb+218
kssdel+228
ksudel_int+280
ksudel+68
kxfpdqs+284
kxfprdp_int+4566
ksbdispatch+367
opirip+522
opidrv+581
sou2o+165
opimai_real+173
ssthrdmain+417
main+256
```

Shared pool memory release due to **tearing down PX sessions.**

Shared Pool Latch Contention – V$PX_SESSIONS

```
SQL> select px.sid sid, s.sql_id
     from v$px_session px, v$session s
     where px.saddr = s.saddr;

    SID SQL_ID
------- -------------
    105 505a4v8cyx05c
    278 505a4v8cyx05c
    807 505a4v8cyx05c
     57 505a4v8cyx05c
    374 505a4v8cyx05c
   2920 505a4v8cyx05c
    534 505a4v8cyx05c
   1941 505a4v8cyx05c
   1336 505a4v8cyx05c
   2424 505a4v8cyx05c
    546 505a4v8cyx05c
   2743 505a4v8cyx05c
    771 505a4v8cyx05c
   2947 505a4v8cyx05c
   1088 505a4v8cyx05c
   ...

38 rows selected.
```

Manual sampling of V$PX_SESSIONS showed this pattern:

SIDs changing rapidly, but SQL_ID always **505a4v8cyx05c**

More PX sessions than defined by **parallel_max_servers=32**
=> PX downgrades

PX session allocation and deallocation thrashes the shared pool!

Shared Pool Latch Contention – SQL 505a4v8cyx05c Execution Plan

```
SQL> select * from table(dbms_xplan.display_cursor('505a4v8cyx05c', format=>'ADVANCED'));

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------------------------------------------------------------
SQL_ID  505a4v8cyx05c, child number 0
-------------------------------------
select childparam0_.parent_id as parent_id6_67_1_, childparam0_.id as
...

Plan hash value: 3055349448

-------------------------------------------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                                   | Name                  | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |    TQ  |IN-OUT| PQ Distrib |
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                            |                       |       |       |     9 (100)|          |       |       |        |      |            |
|   1 |  PX COORDINATOR                             |                       |       |       |            |          |       |       |        |      |            |
|   2 |   PX SEND QC (RANDOM)                       | :TQ10000              |     6 |  1236 |     9   (0)| 00:00:01 |       |       |  Q1,00 | P->S | QC (RAND)  |
|   3 |    INLIST ITERATOR                          |                       |       |       |            |          |       |       |  Q1,00 | PCWC |            |
|   4 |     PX PARTITION HASH ITERATOR              |                       |     6 |  1236 |     9   (0)| 00:00:01 |KEY(I) |KEY(I) |  Q1,00 | PCWC |            |
|   5 |      TABLE ACCESS BY GLOBAL INDEX ROWID BATCHED| PARAMETERVALUE     |     6 |  1236 |     9   (0)| 00:00:01 | ROWID | ROWID |  Q1,00 | PCWP |            |
|*  6 |       INDEX RANGE SCAN                      | IX_PARAMVAL_PARENT_ID |     6 |       |     5   (0)| 00:00:01 |KEY(I) |KEY(I) |  Q1,00 | PCWP |            |
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

Optimizer estimates 6 rows only. Does this really have to run in parallel?

Shared Pool Latch Contention – SQL Execution Statistics (AWR)

SQL ordered by Elapsed Time

- Resources reported for PL/SQL code includes the resources used by all SQL statements called by the code.
- % Total DB Time is the Elapsed Time of the SQL statement divided into the Total Database Time multiplied by 100
- %Total - Elapsed Time as a percentage of Total DB time
- %CPU - CPU Time as a percentage of Elapsed Time
- %IO - User I/O Time as a percentage of Elapsed Time
- Captured SQL account for 72.5% of Total DB Time (s): 14,686
- Captured PL/SQL account for 8.9% of Total DB Time (s): 14,686

Elapsed Time (s)	Executions	Elapsed Time per Exec (s)	%Total	%CPU	%IO	SQL Id	SQL Module	PDB Name	SQL Text
5,469.92	19,258	0.28	37.25	1.27	0.47	505a4v8cyx05c	JDBC Thin Client	PHDM2	select childparam0_.parent_id ...
848.76	2	424.38	5.78	79.91	5.95	8uxwv2vhn53p1	SQL*Plus	PHDM2	BEGIN bto_ai.ai_cleanup_pv_old...

This query has >19,000 executions and takes < 0.3 sec to complete.

This is not a good candidate to run in parallel!

Why is it still running in parallel then?

Shared Pool Latch Contention – Index Degree Of Parallelism (DOP)

```
SQL> select index_name, degree
     from dba_indexes where owner = '&&owner'
     and (degree = 'DEFAULT' or degree > 1);


 INDEX_NAME                      DEGREE
 ------------------------------ ----------
 UQ_PARAMETERVALUE               16
 IX_PARAMETERVALUE_VALUE_NAME    16
 PK_PARAMETERVALUE               16
 IX_PARAMVAL_PARENT_ID           16
```

```
SQL> alter index &&owner.IX_PARAMVAL_PARENT_ID parallel 1;
```

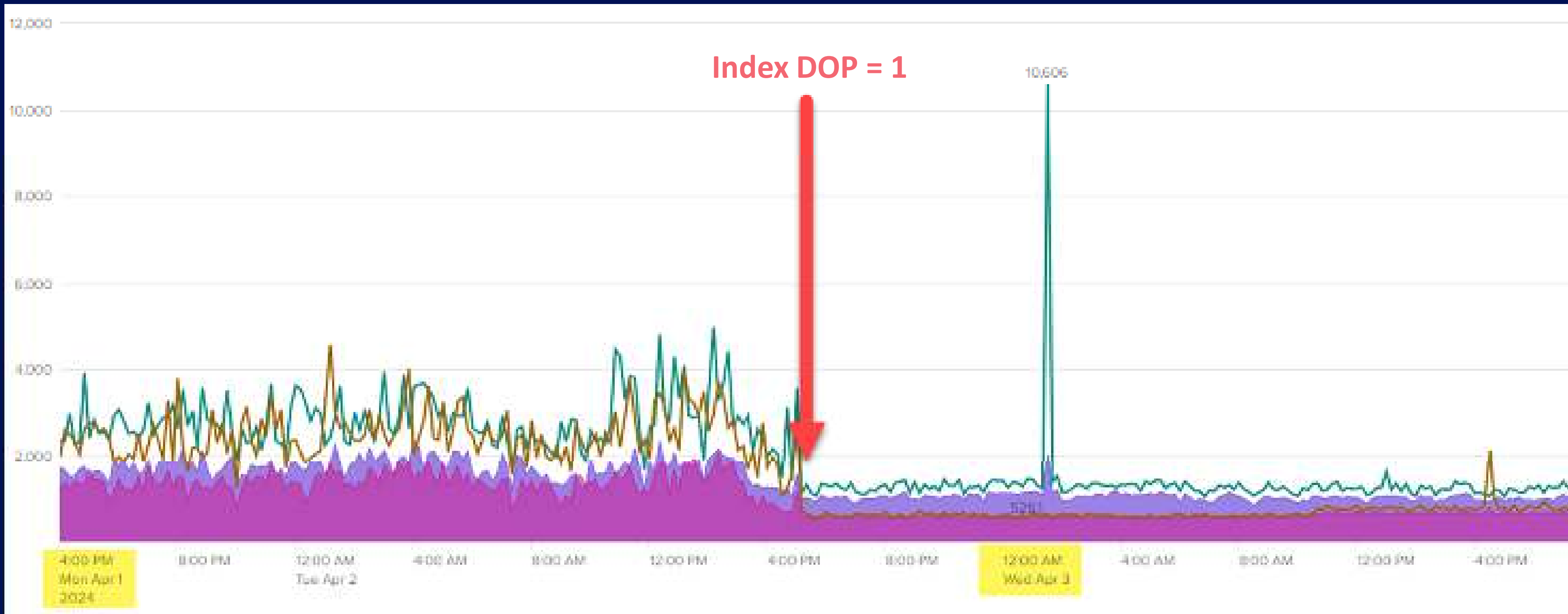
Index DOP of 16.

Reason is unknown (vendor default?).

The query should run in serial!

Moral of the story:
parallel != better or faster

Shared Pool Latch Contention – A Picture tells a thousand words …



Why was the Connection Test slowed down?

```
pid: 244124
        kslgetl+0
        kghalo+8764
        kss_init_private_so_cache+134
        kss_init_proc+128
        ksucrp+1114
        opiino+1394
        opiodr+1253
        opidrv+1094
        sou2o+165
        opimai_real+422
        ssthrdmain+417
        main+256
```

Session Creation Code Path

kslgetl Acquire the shared pool latch in X mode.

kghalo Allocate shared pool memory. This requires an exclusive shared pool latch get.

kss_init Session initialization; create new State Objects (SO) that require shared pool memory.

ksucrp Create and initialize a new process.

These shared pool latch waits are not exposed in v$session or in ASH, because no session exists yet!

Shared Pool Latch Contention – Latch Structure Memory Layout

```
SQL>  select '0x'||trim(0 from addr) laddr
      from v$latch_children where name = 'shared pool'
      and rownum = 1;

 LADDR
 ----------
 0x604746D8
```

Addresses of the latch structures are exposed in v$latch and v$latch_children. We can directly access these with non-Oracle tools ... :-)

Latch free/unused

```
 (gdb) x/6xw 0x604746D8
0x604746d8:  0x00000000  0x00000000  0x019457d9  0x3700026b  0x0000000a  0x0000189a
```

Latch held (in X mode)

When a latch is held in eXclusive mode, the Oracle pid (v$process.pid) is stored in the first 8-byte word of the latch structure.

```
 (gdb) x/6xw 0x604746D8
0x604746d8:  0x00000000   0x0000006c  0x019457d9  0x3700 026b   0x0000000a   0x0000189a
                              pid^^         gets       ?? latch#      level^     location
```

DEMO

Caveat – Noisy Neighbors

Latches are **CDB-level** structures

In a **Multitenant** environment, rogue PDBs will negatively impact other PDBs!

Case 2:
PGA Memory Leak

Starting Situation

PGA Memory Issues – Diagnostics

PGA Diagnostics

- Automatic Trace and Incident Dumps
- PGA Heapdumps
- Process Memory Trace
- Active Session History (ASH)
- Custom Tracing (bpftrace)

ORA-4036 – Oracle Diagnostics: Process Trace File

```
=======================================
PRIVATE MEMORY SUMMARY FOR THIS PROCESS
---------------------------------------
******************************************************
PRIVATE HEAP SUMMARY DUMP
699 MB total:
  697 MB commented, 1181 KB permanent
  561 KB free (192 KB in empty extents),
    696 MB,   1 heap:    "session heap   "            64 KB free held
------------------------------------------------------
Summary of subheaps at depth 1
697 MB total:
  694 MB commented, 149 KB permanent
 2245 KB free (44 KB in empty extents),
    695 MB,  14 heaps:   "koh-kghu sessi "            2099 KB free held
------------------------------------------------------
Summary of subheaps at depth 2
693 MB total:
  689 MB commented, 102 KB permanent
 4077 KB free (0 KB in empty extents),
    344 MB, 22034 chunks:  "PLSQL Collection Bind     " 2012 KB free held
    344 MB, 22033 chunks:  "PLSQL Collection Bind Poin" 2012 KB free held
******************************************************
```

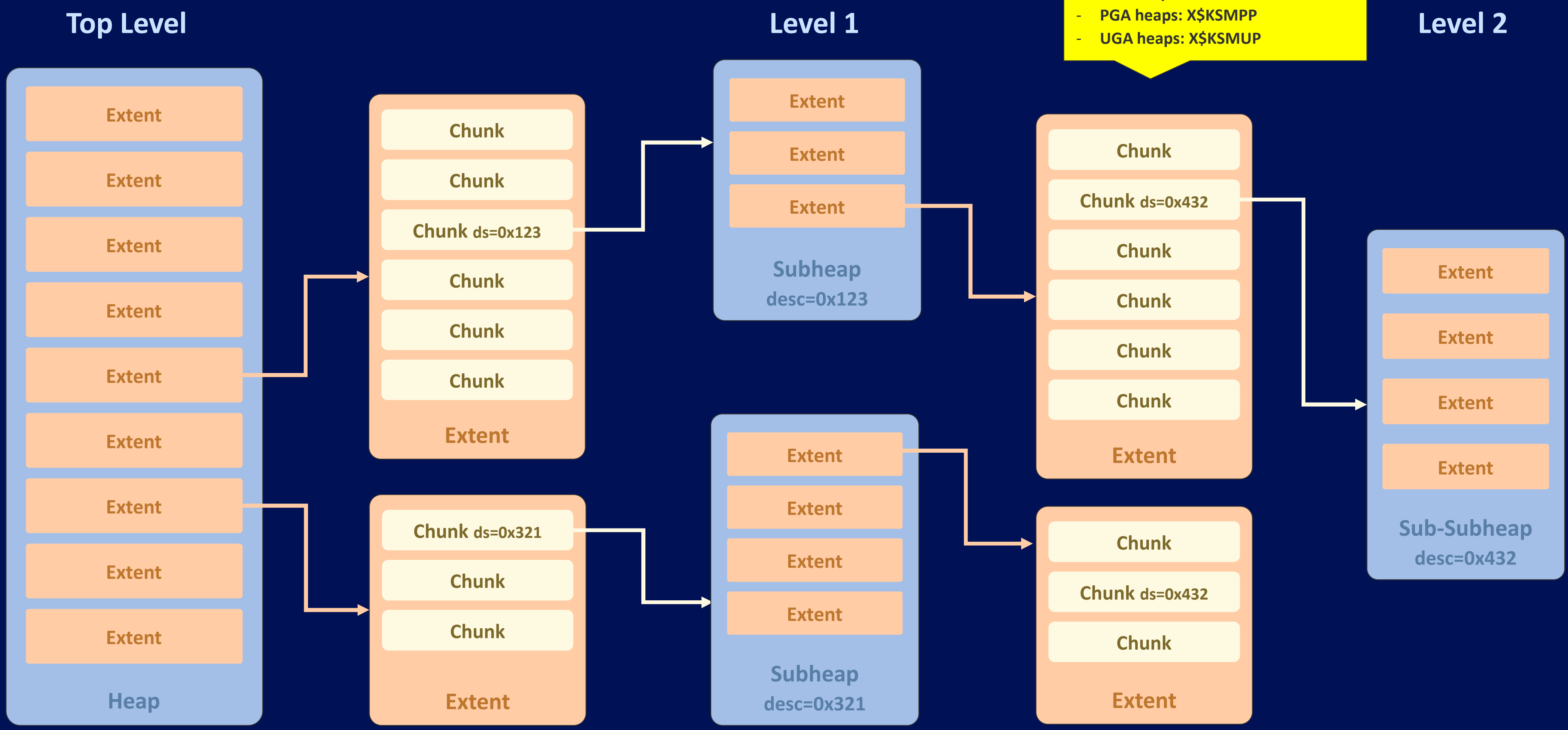
Automatically created when an ORA-4036 occurs.

KOH = Kernel Objects Heap

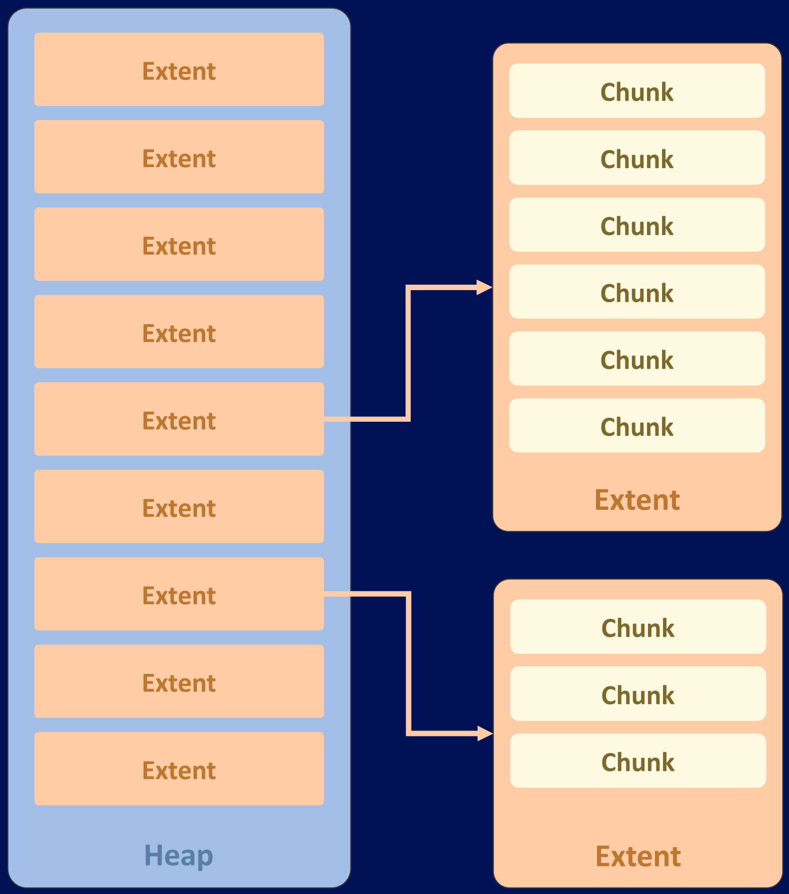
Oracle KGH – Kernel Generic Heap Allocator

The KGH allocator is used for SGA and PGA memory allocations.
Heap and chunk details are exposed via the following x$ tables:

- SGA heaps: X$KSMSP
- PGA heaps: X$KSMPP
- UGA heaps: X$KSMUP



Oracle KGH – Chunk Classes & Descriptions



Chunk Classes:

Freeable chunks can only be freed via the object that allocated them. That is, the KGH memory manager cannot arbitrarily free "freeable" chunks under memory pressure (e.g. a SQLA can only be freed via KGL callbacks).

free
Chunk is free and can be used (chunk on freelist).

freeable
Chunk is in-use, but can be released (chunk not on LRU list).

recreatable
Chunk is in-use, but can be removed and reconstructed if needed (unpinned recreatable chunks are on LRU list).

permanent
Chunks in this state will never be released.

Chunk Descriptions:

Most chunks are associated with a description / comment that provides additional context information on what a chunk is used for.

Memory Leaks – Key Question

Who is allocating **what** over **time?**

Context:
- session id
- username
- sql id
- module / action

Context:
- heaps
- (sub-)subheaps
- chunks
- allocation size

Context:
- sudden and bursty increase?
- slow and steady increase?

PGA Heap Dumps – Examples

Level 1

PGA summary - dump all PGA top level private heaps: PGA, UGA and call heap

```
SQL> oradebug dump heapdump 1
```

Level 0x20000001 (decimal 536870913 = 536870912 + 1)

Private memory dump of top PGA heap + 2 levels of subheap dump recursion

```
SQL> oradebug dump heapdump 536870913
```

Level 0x20000005 (decimal 536870917 = 536870912 + 1 + 4)

Private memory dump of top PGA and UGA heaps + 2 levels of subheap dump recursion

```
SQL> oradebug dump heapdump 536870917
```

PGA Heap Dumps – TPT Heapdump Analyzer

```
-- Heapdump Analyzer v1.03 by Tanel Poder ( https://blog.tanelpoder.com )

  Total_size #Chunks  Chunk_size,        From_heap,       Chunk_type,  Alloc_reason
 ------------ ------- ------------ ----------------- ----------------- -----------------
   337718304   20724        16296 ,   koh-kghu sessi,          freeable,  PLSQL Collectio
   241155920     230      1048504 ,     session heap,          freeable,  koh-kghu sessi
   138411240      33      4194280 ,     top uga heap,          freeable,  session heap
   104856400      50      2097128 ,     top uga heap,          freeable,  session heap
    52427600      50      1048552 ,     top uga heap,          freeable,  session heap
    46134528      44      1048512 ,     session heap,          freeable,  koh-kghu sessi
    26213200      50       524264 ,     top uga heap,          freeable,  session heap
    20444736      39       524224 ,     session heap,          freeable,  koh-kghu sessi
    13106000      50       262120 ,     top uga heap,          freeable,  session heap
     7600320      29       262080 ,     session heap,          freeable,  koh-kghu sessi
     6552400      50       131048 ,     top uga heap,          freeable,  session heap
     5766376      11       524216 ,     session heap,          freeable,  koh-kghu sessi
         […]
```

Script source: Tanel Poder, tpt-oracle Github Repository, Script "heapdump_analyzer"

PGA Heap Dump Analysis – Summary

Who?

Must know the rogue session upfront!

What?

Detailed break-down at chunk level.

Post-processing needed!

Time

Just a snapshot in time!

No history!

Process Memory Trace – Examples

Trace Commands

To clear the trace data, use the PGA_DETAIL_CANCEL trace event.

```
alter session set events
 'immediate trace name PGA_DETAIL_GET level <v$process.pid>';
```

```
oradebug setospid <ospid>

oradebug unlimit

oradebug dump PGA_DETAIL_GET <v$process.pid>
```

Every trace execution updates the V$PROCESS_MEMORY_DETAIL and you must manually save the trace output as it will get overwritten otherwise!

Process Memory Trace – V$PROCESS_MEMORY_DETAIL

21c+: TIME and SQLID columns are exposed in the v$ view

19c: TIME and SQLID columns are populated in the x$ table in 19.18+, but not exposed in the v$ view!

```
SQL> desc V$PROCESS_MEMORY_DETAIL

Name                     Null?    Type
------------------------ -------- ------------
PID                               NUMBER
SERIAL#                           NUMBER
CATEGORY                          VARCHAR2(15)
NAME                              VARCHAR2(26)
HEAP_NAME                         VARCHAR2(15)
BYTES                             NUMBER
ALLOCATION_COUNT                  NUMBER
HEAP_DESCRIPTOR                   RAW(8)
PARENT_HEAP_DESCRIPTOR            RAW(8)
CON_ID                            NUMBER
```

```
SQL> desc X$KSMPGDSTA

Name                   Null?    Type
---------------------- -------- ------------
ADDR                            RAW(8)
INDX                            NUMBER
INST_ID                         NUMBER
CON_ID                          NUMBER
KSMPGDSTA_PID                   NUMBER
KSMPGDSTA_SER                   NUMBER
KSMPGDSTA_PAFLG                 NUMBER
KSMPGDSTA_TIME                  DATE
KSMPGDSTA_SQLID                 VARCHAR2(13)
KSMPGDSTA_TOTMB                 NUMBER
KSMPGDSTA_COMMENT               VARCHAR2(26)
KSMPGDSTA_CATNAME               VARCHAR2(15)
KSMPGDSTA_HEAPNAME              VARCHAR2(15)
KSMPGDSTA_NUM_ALLOC             NUMBER
KSMPGDSTA_BYTES_ALLOC           NUMBER
KSMPGDSTA_DS                    RAW(8)
KSMPGDSTA_PARENT_DS             RAW(8)
```

Process Memory Trace – Automatic Snapshot Behavior

Important Notes:

- With **fix 21533734**, an automatic snapshot of the fg process memory usage is taken when a process uses **>500 MB of PGA and each 20% growth after that**.
- Fix 21533734 is first included in Oracle **versions 19.18 and 20.1**.
- The behavior can be controlled via the following underscore parameters:
  - _pga_auto_snapshot_percentage (default 20 percent)
  - _pga_auto_snapshot_threshold (default 500 MB)

Source: My Oracle Support: Bug 21533734 - V$process_memory_detail should be automatically populated (Doc ID 21533734.8)

Process Memory Trace – Summary

Who?

Must know rogue session upfront.

New time and sql_id fields in 19.18+.

What?

Detailed break-down at chunk level.

Post-processing needed!

Time

Just a snapshot in time!

No history!

PGA Memory Leaks – Active Session History (ASH)

```
SQL> desc DBA_HIST_ACTIVE_SESS_HIST

Name                           Null?    Type
------------------------------ -------- ------------
SNAP_ID                        NOT NULL NUMBER
DBID                           NOT NULL NUMBER
INSTANCE_NUMBER                NOT NULL NUMBER
SAMPLE_ID                      NOT NULL NUMBER
SAMPLE_TIME                    NOT NULL TIMESTAMP(3)
…
SQL_ID                                  VARCHAR2(13)
SQL_OPNAME                              VARCHAR2(64)
PLSQL_ENTRY_OBJECT_ID                   NUMBER
PLSQL_ENTRY_SUBPROGRAM_ID               NUMBER
PLSQL_OBJECT_ID                         NUMBER
PLSQL_SUBPROGRAM_ID                     NUMBER
[…]
PGA_ALLOCATED                           NUMBER
```

The Active Session History provides a lot of context information and a historical activity record!

PGA Memory Leaks – ASH Example Query (Starting Point)

```
select
  sample_time,
  session_id,
  sql_opname,
  top_level_sql_id,
  sql_id,
  sql_child_number,
  sql_plan_line_id,
  in_parse,
  in_hard_parse,
  in_sql_execution,
  round(pga_allocated/1024/1024,1) pga_mb
from
  dba_hist_active_sess_history
where
    session_id = &&sid
and sample_time between timestamp'&&start_time' and timestamp'&&end_time'
and instance_number = (select instance_number from v$instance)
order by sample_time
/
```

With ASH you can narrow down into statement and execution plan line details!

Active Session History (ASH) – Summary

Who?

Always on.

Drill-down possible.

What?

Shows only total PGA, no detailed break-down!

Time

Historical track record (1 or 10 sec resolution).

Oracle KGH – Memory Allocation Functions

kghalp – Kernel Generic Heap Manager Allocate Permanent chunk

```
kghalp(arg0, arg1, arg2, arg3, arg4, char *comment)
```

kghalf – Kernel Generic Heap Manager Allocate Freeable chunk

```
kghalf(arg0, arg1, arg2, arg3, arg4, char *comment)
```

kghfre – Kernel Generic Heap Manager Free chunk

```
kghfre(…)
```

The KGH functions are called whenever a memory chunk is allocated or freed!

Function Parameters*

- arg0 – 4: Unknown
- arg5: **comment -** Chunk comment

*Function and parameter names source:
Tanel Poder, tpt-oracle, Script "trace_kghal.sh"

Oracle KGH – Tracing Idea

```
uprobe:/u01/app/oracle/product/19.0.0.0/dbhome_1919_1/bin/oracle:kghalp,
uprobe:/u01/app/oracle/product/19.0.0.0/dbhome_1919_1/bin/oracle:kghalf,
/ str(@kspssidst) == "MY_ORACLE_SID" &&
  ( str(arg5) == "PLSQL Collection Bind" || str(arg5) == "PLSQL Collection Bind Pointer") /
{
   @in_trace[tid] = 1;
   @func[tid] = func;
   @reason[tid] = (uint64) arg5;
   $reason = @reason[tid];
   $fsbase = uptr(curtask->thread.fsbase);

   /* x$ksupr offsets */
   $paddr_off = (uint64) 0xff90;     /* tls offset */
   $ksuprpum_off = (uint64) 0xe90;  /* pga used   */
   $ksuprpnam_off = (uint64) 0xe70; /* pga alloc1 */
   $ksuprpram_off = (uint64) 0xe58; /* pga alloc2 */

   /* x$ksupr data */
   $paddr_p = uptr($fsbase - $paddr_off);
   $paddr = *(uint64 *) uptr($paddr_p);
   $pga_used = *(uint64 *) uptr($paddr + $ksuprpum_off);
   $pga_alloc1 = *(uint64 *) uptr($paddr + $ksuprpnam_off);
   $pga_alloc2 = *(uint64 *) uptr($paddr + $ksuprpram_off);
   $pga_alloc = (uint64) ($pga_alloc1 + $pga_alloc2);
   […]
}
```

Trace every kghalp and kghalf call.

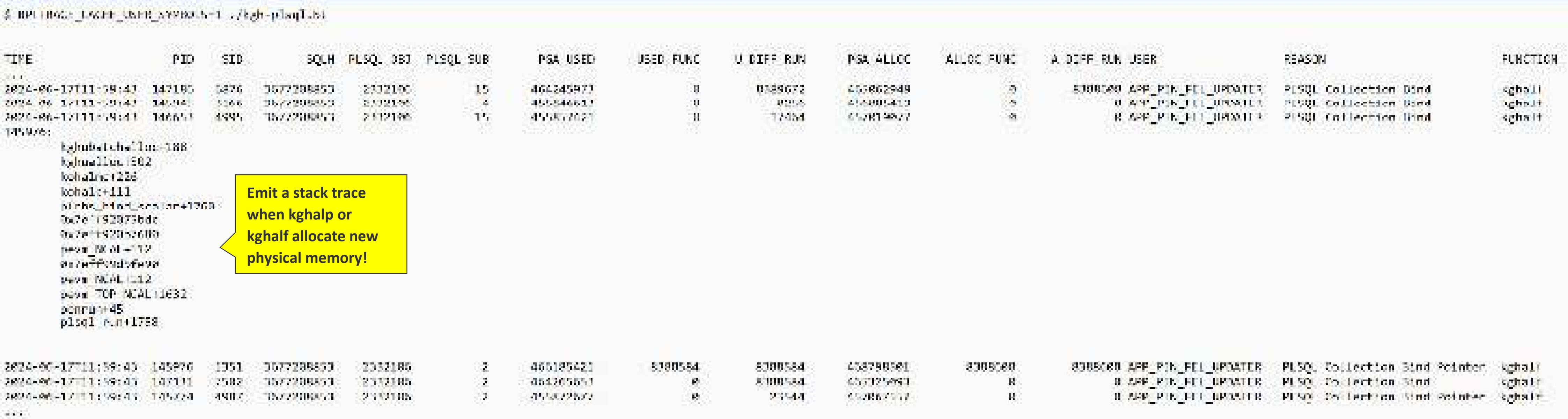
Enrich trace with additional context information from v$session and v$process.

Script idea based on:

Stefan Koehler, soocs-scripts Github Repository, Script dtrace_kghal_pga_code.sh

Tanel Poder, TPT Github Repository, Script trace kghal.sh

Oracle KGH – Trace Output




```
TIME        : Time of allocation
PID         : OS pid
SID         : Session id
SQLH        : SQL hash value
PLSQL_OBJ   : PL/SQL object id
PLSQL_SUB   : PL/SQL subobject id
PGA_USED    : PGA memory currently used
USED_FUNC   : Increase in PGA used mem by function
```

```
U_DIFF_RUN  : PGA used mem runtime difference (since script start)
PGA_ALLOC   : PGA memory currently allocated
ALLOC_FUNC  : Increase in PGA alloc mem by function
A_DIFF_RUN  : PGA alloc mem runtime difference (since script start)
USER        : DB username
REASON      : Chunk allocation reason
FUNCTION    : KGH memory allocation function
```

Script source: [Christoph Lutz, Github Repository, Script kgh-alloc-by-reason.bt]

Oracle KGH – Trace Analysis

```
$ ./kgh-plsql-analyze.sh <kgh_trc_log_file>
```

PID	Sum Total PGA Alloc	Sum Bind Alloc	Sum Bind Pointer Alloc	Bind + Bind Pointer
376402	150994944	150994944	0	150994944
377682	148897792	2097152	146800640	148897792
378123	153092096	153092096	0	153092096
380253	151060480	0	146800640	146800640
380641	150994944	113246208	37748736	150994944
380663	155189248	2097152	153092096	155189248
381090	155189248	142606336	12582912	155189248
381639	153288704	0	153092096	153092096
381984	153092096	0	153092096	153092096
382089	148897792	0	148897792	148897792

- We can write custom tools to summarize and aggregate the trace output
- The example above shows which PIDs have allocated the most "PL/SQL Collection Bind" chunks during the measurement interval (top 10)

DEMO

Custom Tracing (bpftrace) – Summary

Who?

Always on.

Capture all context details.

What?

Record allocation of every single chunk.

Time

Log changes over time.

Full history!

Case 3:
ACS, Bind Peeking and Plan Flips

Historical Context

9i

Bind Variable Peeking

"Peek at" the bind variables during hard parse and compile a plan using the selectivities of the "peeked" binds.

This still happens with 19c!

10g

Stats Collection with Automatic Histogram Creation

Automatic gathering of histograms made bad situations caused by Bind Variable Peeking even worse and much more unpredictable.

11g

Adaptive Cursor Sharing (ACS)

Dynamically adapt execution plans at runtime based on the selectivity of values used in bind variables.

The Evergreen … (literally happens monthly)

Call from App Owner: We have a huuuuge performance problem!

DBA: Ah ok, do you have a timestamp to narrow it down?

App Owner: Ehm, it's constantly bad since yesterday evening!

DBA: Ehm, did you change something?

AppOwner: Äh, no – not to my knowledge …

DBA: Ok, let me check!

So, what's going on?

```
...Extract

SQL> @aw 1=1

Showing top SQL and wait classes of last minute from ASH...

     Total
   Seconds      AAS %This   SQL_ID        SESSION                                 WAITCLASS       FIRST_SEEN          LAST_SEEN
 --------- ------- ------- ------------- --------------------------------------- --------------- ------------------- -------------------
        42    17.5    71%   846wumz55pycz WAITING                                 User I/O        2024-07-02 09:59:09 2024-07-02 09:59:59
        17     1.2    13%   6kfjvu1dfqr3x WAITING                                 Concurrency     2024-07-02 09:59:10 2024-07-02 09:59:31
```

Spot On! 846wumz55pycz → 17.5 active sessions on average in the last minute

Was there a plan flip?

```
...Extract

CDB1.PDB1 SQL> @awr/awr_sqlstats_per_exec 846wumz55pycz % sysdate-7 sysdate

BEGIN_INTERVAL_TIME SQL_ID        PLAN_HASH_VALUE EXECUTIONS ELA_MS_PER_EXEC CPU_MS_PER_EXEC ROWS_PER_EXEC LIOS_PER_EXEC BLKRD_PER_EXEC
------------------- ------------- --------------- ---------- --------------- --------------- ------------- ------------- --------------
2024-06-30 17:56:22 846wumz55pycz      1176963889        141               1               0           1.0            37              0
2024-06-30 18:56:28 846wumz55pycz      1176963889        213               2               0           1.0            38              0
2024-07-01 19:56:25 846wumz55pycz      1176963889         75               1               0           1.0            37              0
2024-07-01 20:26:19 846wumz55pycz      1353868891        357            5804            5504           1.0        341220         341182
2024-07-01 21:26:45 846wumz55pycz      1353868891        101            5798            5409           1.0        344587         344553
2024-07-01 22:56:54 846wumz55pycz      1353868891        198            5643            5369           1.0        345987         345952
```

Plan flips here – from 1-2ms to almost 6sec per execution!

Why did the Plan flip (assuming stats are fresh)?

check the query, or rather the table(s) and predicates involved:

```
SQL> @sqlid 846wumz55pycz %

Show SQL text, child cursors and execution stats for SQLID 846wumz55pycz child %

HASH_VALUE PLAN_HASH_VALUE   CH# SQL_TEXT
---------- --------------- ----- ------------------------------------------------------------------------------------------
968766335       1353868891     0 select .... from SUBSCRIPTIONS ...join ... left outer join ... where A_ACCOUNTNUMBER = :1
```

check data distribution of the column in the where clause of the table being queried:

```
SQL> select * from (select A_ACCOUNTNUMBER, count(*) from SUBSCRIPTIONS group by A_ACCOUNTNUMBER order by 2 desc) where rownum <= 10;

A_ACCOUNTNUMBER                       COUNT(*)
------------------------------------ ----------
1683019842                              162420
1955701227                              106553
1140847506                               17223
1502741410                                3625 --from here
1684390416                                2688
1575064115                                2506
1167065059                                1042
1501588712                                1030
1448184576                                 970
1996574562                                 830 --to here it's more or less within the same range(bucket) and above it jumps
```

Ingredients for plan flips given!

check peeked binds:

```
SQL> @xia 846wumz55pycz %

Peeked Binds (identified by position):
--------------------------------------

   1 - :1 (NUMBER): 1955701227
```

The optimizer peeked a value which is not requested so often, and values in lower ranges which are requested way more often, result in a bad plan.

Chasing (Peeked) Binds

Parse Time

- DBMS_XPLAN option **"+PEEKED BINDS"** only shows the initial peeked bind value used on **hard parse**

Execution / Runtime:

- "Runtime bind values" can be found in:
  - SQL Monitor Report, which is only created if db time of query >5 sec or query is using PX
  - SQL Trace: full capture (needs to be enabled)

Bear in mind that **V$SQL_BIND_CAPTURE** only captures binds in the following situations:

- During a hard parse
- A soft parse that creates a new child cursor
- If the last captures was "_cursor_bind_capture_interval" seconds or longer ago (default: 900 sec)
- Column type is not "LONG" or "LOB"

Further down the Road...

```
CDB1.PDB1 SQL> select is_bind_aware,is_bind_sensitive from v$sql where sql_id='846wumz55pycz ';

IS_BIND_AWARE IS_BIND_SENSITIVE
------------- -----------------
N             Y
```

Why is this SQL not bind aware?

- Queries with bind variables in predicates are generally marked bind sensitive (depends on whether or not the bind variable is a collection).
- ACS *might* mark a query bind aware if bind variable values significantly affect the number of rows processed. Precon for this is parsing. (In turn - if the application keeps the cursor open, the sql engine won't be able to generate a new child cursor for a better plan)
- A SQL is monitored, if certain criterias are given, the cursor will become "BIND_AWARE".

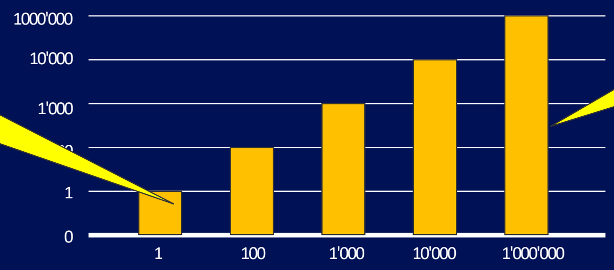
```
SQL> select sql_id,child_number,bucket_id,count from v$sql_cs_histogram where sql_id='846wumz55pycz';

SQL_ID                                  CHILD_NUMBER  BUCKET_ID      COUNT
--------------------------------------- ------------ ---------- ----------
846wumz55pycz                                      1          0       3616
846wumz55pycz                                      1          1         36
846wumz55pycz                                      1          2          0
846wumz55pycz                                      0          0          2
846wumz55pycz                                      0          1          0
846wumz55pycz                                      0          2
```

This view is used by the sql engine to determine whether or not a cursor is made bind aware

...but – the bucket threshold calculations are a) undocumented and b) dirty

Data Skew + Histograms + Bind Variables – What to do?!

SELECT count(*) FROM t WHERE n = 1;
=> Index Range Scan



SELECT count(*) FROM t WHERE n = 1000000;
Full Table Scan

SELECT count(*) FROM t WHERE n1 = :n

You're facing a situation with …

- Query with bind variables
- Data skew
- Histograms
- Bad performance and users complain

… and you have now clue about

- Data constellation in the future
- Future database design changes
- Future access patterns

This is first and foremost an application problem (not that anybody wants to hear that …)!

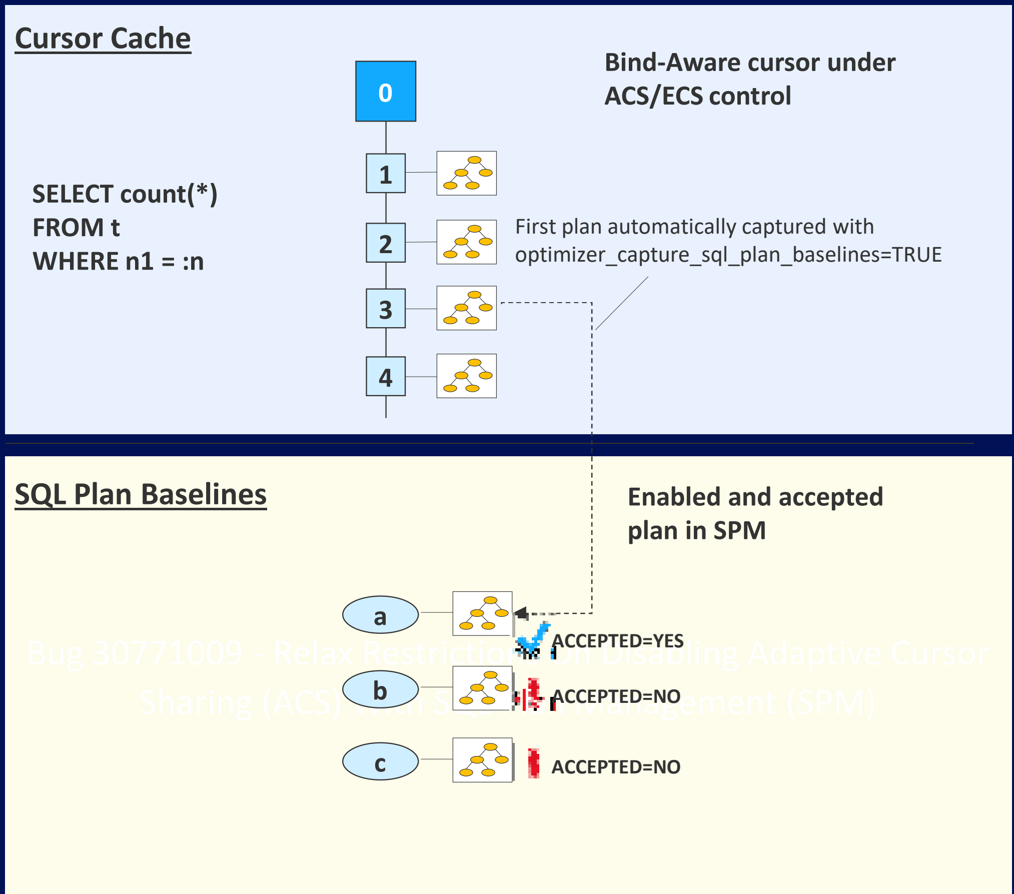
What can / should you do??

- **Data skew, histograms and bind variables don't mix!**
- We have very limited means at our disposal with which we can only address the "now" part of the problem to some degree.
- Hotfixing this kind of situation is almost guaranteed to call for trouble further down the road sometime in the future (the "later" part).
- Tactically, you may inject the **BIND_AWARE** hint with a **SQL Patch** into queries that are known to suffer from poor or flipping plans (risk: this can lead to "high version count" issues) → can't have it all...

```
SQL> @create_sql_patch 846wumz55pycz BIND_AWARE
```

- SQL Plan Management (SPM) / SQL Plan Baselines can also help but there is a pitfall you must be aware of – otherwise you run the risk of making things even worse (s. next section)!

How do SPM and ACS interact with each other?



SPM Automatic Plan Capture and ACS

Only the first plan is captured and ACCEPTED!

Additional Plans are added, but not ACCEPTED!

SPM Plan Selection and ACS

If a Baseline has only one accepted plan for a statement, then ACS is automatically disabled for that statement!

Restrictions:

Beware of Bug 30771009 - Relax Restrictions On Disabling Adaptive Cursor Sharing (ACS) With SQL Plan Management (SPM)

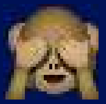
SQL Containing More Than 8 Bind Variables is not Marked as Bind Sensitive (Doc ID 1983132.1) → fix control: 33627879

If a query is executed from within PL/SQL ACS might not work as expected due to internal caching mechanos

If an application keeps a cursor open, ACS will not kick in, since it requires parse calls

Hold on – How did we fix the Problem?

After sharing the findings with the application guys, they spotted that the "top 3" accountnumbers were produced by their monitoring tool.



```
SQL> select * from (select A_ACCOUNTNUMBER, count(*) from SUBSCRIPTIONS group by A_ACCOUNTNUMBER order by 2 desc) where rownum <= 10;

A_ACCOUNTNUMBER                      COUNT(*)
------------------------------------ ----------
1683019842                           162420
1955701227                           106553
1140847506                            17223
1502741410                             3625
1684390416                             2688
1575064115                             2506
1167065059                             1042
1501588712                             1030
1448184576                              970
1996574562                              830
```

"artificial" data leading to plan flip

Hotfix: inject a sql patch with bind_aware hint to force adaptive cursor sharing.

Two weeks later: the 3 accountnumbers where deleted, we removed the patch and the plan stabilized.

Conclusion

Once you've migrated to Autonomous Cloud and enabled the underscore parameter

_ai_fix_all_my_problems=true

you will no longer have such issues!

Happy Troubleshooting!

Questions, feedback, comments?
We look forward to hearing from you!

christoph.lutz@swisscom.com
@chris_skyflier
Symposium 42 | @sym_42

thomas.mayer@swisscom.com

References

References

Andrey Nikolaev, Latch internals, RUOUG Seminar, 2012-12-06

Brendan Gregg, Flame Graphs, 2020-10-31

Christoph Lutz, Oracle Github Repository, Script kgh-alloc-by-reason.bt

Christoph Lutz, Oracle Github Repository, Script kslgetl-shared-pool-stacks.bt

Mohamed Houri, Adaptive Cursor Sharing, Short answer to sharing cursors and optimizing SQL, 2015-11-28

Oracle Corp, SQL Plan Management in Oracle Database 19c, Whitepaper, 2019-03-13

Stefan Koehler, soocs-scripts Github Repository, Script dtrace_kghal_pga_code.sh

Tanel Poder, TPT Github Repository, Script ashtop.sql

Tanel Poder, TPT Github Repository, Script dashtop.sql

Tanel Poder, TPT Github Repository, Script latchprofx.sql

Tanel Poder, TPT Github Repository, Script trace_kghal.sh