RESOLVING CHILD CURSOR ISSUES RESULTING IN MUTEX WAITS

Martin Klier, DBA
Klug GmbH integrierte Systeme
Teunz

Keywords

Oracle Database, Adaptive Cursor Sharing, Shared Pool, Library Cache, Mutex, Wait Event

Introduction

"Resolving Child Cursor Issues Resulting In Mutex Waits" is intended to be an information for Oracle DBAs and DB developers concerned about cursor-related serialization wait events, with a focus on Oracle RDBMS Version 11gR2.

The document contains some basics as parsing and the nature of serialization, some corresponding wait events as well as examples, reproduction of common issues and their explanation. Furthermore, it provides links to further readings, as MOS papers and other author's work.

This white paper and its presentation were written in early 2012 from scratch, initially for the IOUG COLLABORATE 12.

Oracle parsing and Child Cursors

SQL coming into the DB has to be validated, interpreted, compiled, before it can be executed. On this way, the database system performs a Syntax Check, Semantic Check, (Cost Based) Optimization, Row Source Generation and others. We will detail a bit into Shared Pool issues, but covering the whole process is well beyond the scope of this paper. Its basic understanding is kind of a prerequisite for the following parts.

Cursor Sharing

Hard Parse

SQL issued against the database usually is coded into an application, and the parsing session is JDBC, OCI or another connect method. After making sure that syntax (SQL grammar) and semantics (objects, data types...) are fine, the parsing process is going to optimize the access path between various DB objects. In simple cases, it's just the choice between "index" and "table access full", in more complicated situations, it's a combination of more than one table and their corresponding indexes. The DB performs a kind of brute-force approach to this problem, and classifies all results by their artificial "cost" factor. Despite the fact that this "cost-based optimizer" has a number of shortcuts, you can imagine that creating an execution plan may be a lengthy and expensive journey. In this context, "expensive" means consuming CPU resources and thus, keeping others from doing their work. This procedure, every SQL statement has to go through at least once in its life, usually is referred as a *hard parse*.

Parent- and Child Cursor

Knowing these facts very well, someday Oracle started looking for a way to avoid this effort as often as possible. To achieve the goal, they introduced the feature of *Cursor Sharing*: Databases using Cursor Sharing will store all parsed SQL in the Library Cache section of Shared Pool. It's named a *Cursor* now: We distinct between a *Parent Cursor*, which contains the static metadata like the SQL full text, and a *Child Cursor*, with all dynamic variables.

Note: If we have the same SQL, but different metadata, there will be more children for one Parent.

Shared Pool Check

The Library Cache is organized as a hash tree, based on the hash value of the literal SQL text coming from the DB user interface. As soon as a SQL went successfully through syntax- and semantic check, the parser performs the *Shared Pool Check*, that looks up this hash tree structure, to verify if a SQL with the same text is already stored there. If it isn't, the hard parse described above will take place, and afterwards a new parent/child cursor pair is inserted into the Shared Pool / Library Cache.

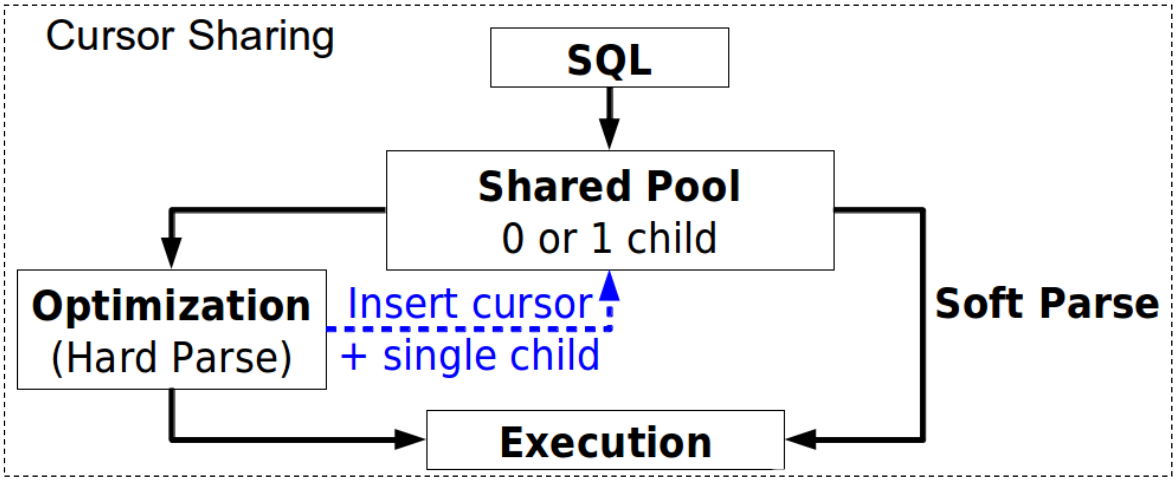

Figure 1: Cursor Sharing

Soft Parse

But if a matching cursor definition is already there, the DB happily re-uses the existing cursors and execution plans to save time and machine load. This is, what performance analysts are calling a “soft parse”, and what we nearly always want to have.

Bind Variables

To be able to benefit from Soft Parsing, despite the fact that all of us have to issue very similar but not identical SQL statements, (think of "SELECT something FROM table WHERE anything=13"), Oracle introduced *Bind Variables*. Bind Variables are visible in : notation in the SQL source code, and can be loaded with real-world values by the application:

```
SELECT something FROM table WHERE anything=:X;
```

So the hash value remains the same while "anything" changes, and so we avoid unnecessary Hard Parsing.

Disadvantage of Cursor Sharing

In dynamic environments, the Cursor Sharing with bind variables from last section has a downside. Let's assume, for the first time Oracle is parsing our example SQL, "anything=:X (=42)" is given. In our table, there are millions of entries with "42" in the "anything" column, and so the selectivity of our predicate is nearly zero. The execution plan will be made serving this demand optimally. Usually, this means a full table scan. The execution plan and the corresponding cursor is stored in Shared Pool, indexed by the hash value of the SQL sentence.

Now, the example SQL is parsed a second time, now with predicate "anything=:X (=43)". For this filter, there is one single row matching. Despite this fact, since the hash value of SQL is the same, Oracle will decide to Soft Parse, and re-use the plan initially made for no selectivity. So the DB in the next step will perform a full table scan, which means unnecessary buffer gets, if not physical reads, but in any case, severe loss of time. A smart index access would have been way better here.

Adaptive Cursor Sharing (11g and above)

Introduced in 11gR1, Adaptive Cursor Sharing tried to reduce the impact of the selectivity problem described in the last section. Before 11g, new child cursors have been created for lots of reasons, but never in order to learn from a wrong selectivity decision.

Cardinality Feedback

The 11g Cardinality Feedback can mark a Parent Cursor, if the result set does not match the expectations being an outcome of the cost-based optimization process. Should a Parent Cursor be marked this way, and next Hard Parse gives us another plan, a new Child Cursor is created and inserted under the old Parent.

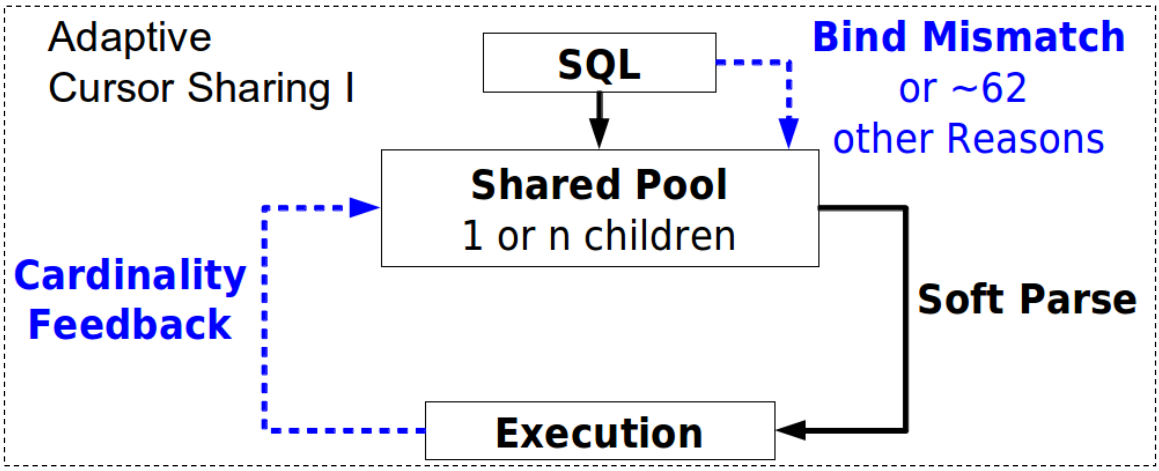

Figure 2: Adaptive Cursor Sharing I

There are two modes, so the Parent Cursor can be marked as

- "bind sensitive", if the optimizer wants the cursor to be re-parsed in order to find out if maybe there's a better plan
- "bind aware", as soon as there's more than one child cursor for selectivity reasons. This marker is paired with a bind set information.

Hard-, Soft-, and Harder Soft Parse

After a parent cursor had been marked as "bind aware", a next parsing of this SQL can't be a classical Soft Parse any more. Now the optimizer has to decide, which of the existing execution plan / child cursor pairs is the best for the bind selectivity given by the user. This is no Hard Parse, and this is no Soft Parse. It's a kind of *"Harder" Soft Parse*, examining the bind information given for each cursor, and deciding which existing child/plan should be reused.

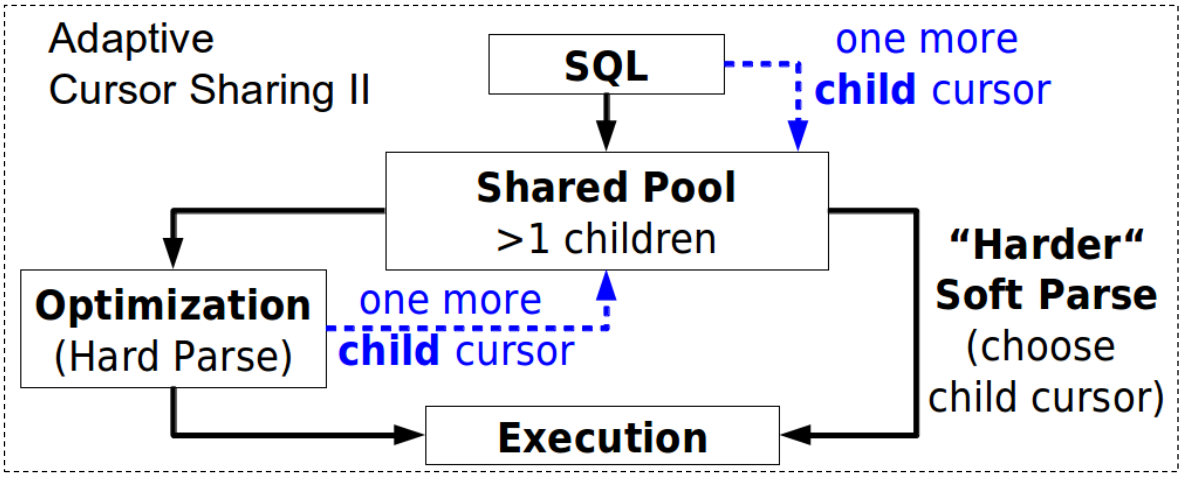

Figure 3: Adaptive Cursor Sharing II

So with Harder Soft Parsing, we do not only "peek" into the bind variable at initial parse time, but at every time a known SQL comes into the parsing process. Of course, this and the comparison with the cursors already being in the Library Cache, needs resources.

Cursor Creation and Invalidation

Oracle Documentation / MOS is explaining 64 reasons, why a new cursor might be inserted or an old one may become invalid. This paper only lists a summary of the few most important ones.

Optimizer environment

If the optimizer mode changes (ALL_ROWS vs. FIRST_ROWS), it is most likely, that a new execution plan has to be created. So invalidating cursors here is necessary to fulfill the user's expectations.

Outline mismatch

The same is true for Outline Mismatch. If another outline is given for the new environment where the SQL runs in, an existing cursor may be absolutely wrong, especially its execution plan.

NLS- and user identity issues

If we are issuing SQL from different NLS environments, different sorting or filtering may be wanted, and thus it's not granted that all objects can be used in the way as they were before. An example: Function based indexes.

Furthermore, if we are parsing the same SQL from a different schema where objects have the same names by coincidence, cursors coming from different semantics are absolutely unsuitable. Oracle just has to insert new ones. The decision is made by looking at the parsing schema.

Cardinality Feedback

We have been on this topic recently, in the last major section. If we decide to have another execution plan for a bind set, either a new cursor has to be created, taking it into account, or in case it already exists, the child cursor hosting the old plan will be invalidated and replaced.

Bind mismatch

Bind Mismatch is the most unwanted reason, as we will prove in later sections. Bind mismatch means that something about the bind variables was different. The list is long, but length migration and datatype difference are the most common ones.

Mutexes

Serialization

Even worse than using CPU resources for the plain memory operations, accessing or changing existing cursors in RAM needs serialization – nobody wants his own memory structure manipulated while reading or writing it. As an example, let's have a look at a singly linked list, a classic in computer science.

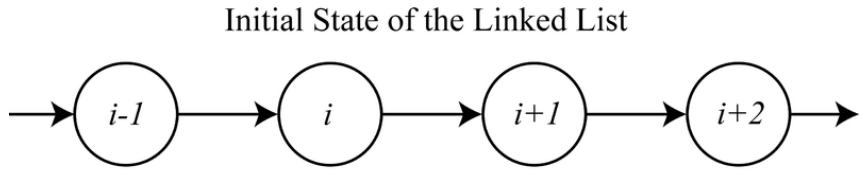

Figure 4: Linked List - Initial State

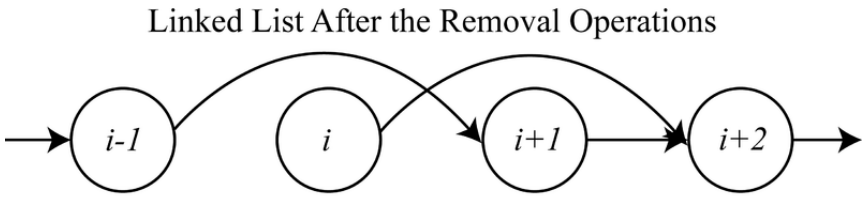

Figure 5: Linked List - After Removal

Node i-1 is linked to node i, which is connected to node i+1. If we want to remove nodes i and i+1, we have to unlink them from their predecessor.

Trouble is, that if we do the removal simultaneously, even though we are successfully deleting node i by relinking node i-1 to node i+1, we sabotage the deletion of node i+1 in the same operation.

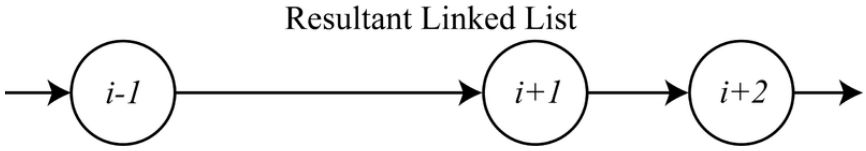

Figure 6: Linked List - Resultant State

Despite the other thread unlinked node i+1, it's not deleted because we immediately link node i-1 to it. Failed - the only way to do this properly, is to do it step by step, and this is what serialization means.

(All Linked List graphics are coming from Wikipedia, Author: KMVisor)

The Structure

A mutex is an abbreviation for “mutual exclusion”, and it means a fine-grained serialization structure. Basically, it works same way as a latch does, but it's lightweight, often directly hardware-supported and thus, *fast*. Thanks to these facts, it's suitable for a wide-spread use.

Starting in 10gR2, Oracle replaces lots of latches by mutexes. Being fast and having a small footprint, a mutex has a double improvement on performance: Speed is one thing, but as well there is less need to share those little structures, so we reduce "false contentions". Latches often have been shared between many cells of a structure; mutexes are usually dedicated exclusively to one logical piece.

In our case this means, that each parent and child cursor has its own mutex, and we do not have to contend for a latch that's co-used by many cursors.

Mutex Contention

Nevertheless, mutex contention still *is* possible, and not as uncommon as many may think. As always, contention comes from parallel execution being serialized: Two processes are applying for the same resource. One holds it, the other one wants it.

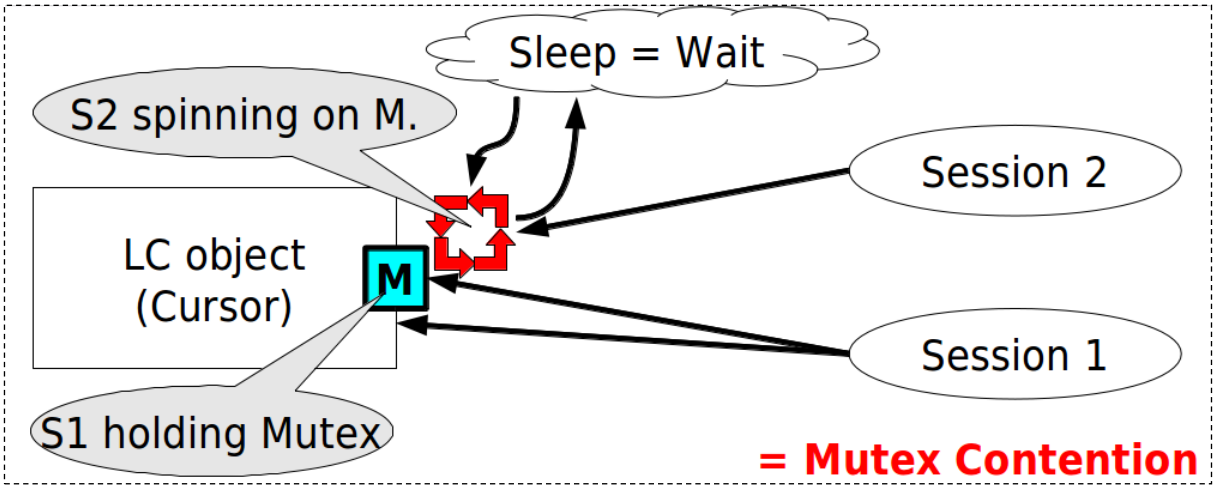

Figure 7: Mutex Contention

In this simplified example, session 1 uses the LC object / cursor in question. Due to that, it also holds the corresponding mutex, that's the responsible serialization structure. Session 2 wants to use the same LC object / cursor in an incompatible way, and thus, *contends* for the mutex protecting it. Repeatedly asking for the status of a mutex is known as "spinning": A tiny piece of memory is queried over and over again, as one can imagine, this intensely consumes CPU. But to allow the OS scheduler and other multitasking mechanics to use this CPU as well from time to time, we have a maximum spin time defined (in Oracle, parameter _spin_count). When this time is out, the process "sleeps" for this mutex. After a while, it comes back and spins again, until the mutex is freed by session 1.

Waits

Especially since 10gR2 and 11g, the Oracle *Wait Interface* is a powerful source of information for performance analysis, and shows information coming from Oracle's code instrumentalization. The Oracle kernel code itself reports those waits, and so we get a quite good impression what is (not) going on at the moment. As explained in last section, contentions are causing active spinning, this means it's loading the CPU without doing real work. As soon as they sleep for a while, Oracle is recording this as a so-called *Wait Event*. This paper focuses on cursor-related mutex wait events, so we will discuss the top five of them in greater detail.

cursor: mutex X

The “X” wait event tells us that an exclusive lock on a cursor-related mutex is happening. Processes are holding this mutex or are contending for it, as soon as they do or want to do:

- Insert a new child cursor under a parent
- Do bind peeking for it (=capture SQL bind data)
- Modify cursor-related statistics

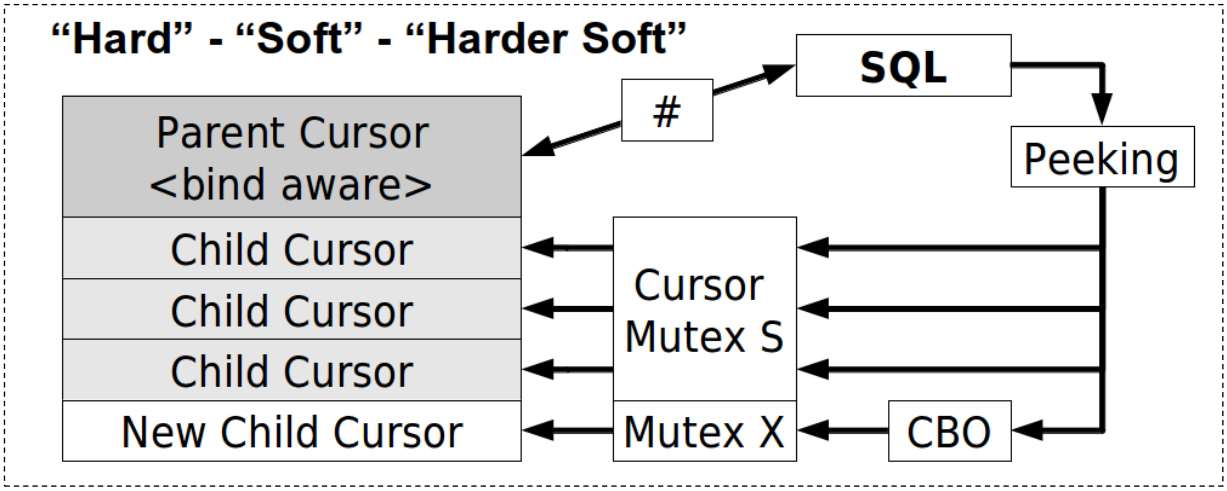

Figure 8: Library Cache Access Serialization

cursor: mutex S

The “S” wait event tells us that a shared lock on a cursor-related mutex is happening. Processes are holding this mutex or are contending for it, as soon as they do or want to do a change in the reference count of the mutex itself. This count indicates the number of processes interested in this mutex. The basic meaning is, “a new guy is interested, and thus, spinning/waiting”

cursor: pin X

The “pin X” wait indicates sleeps on an exclusive mutex protection against deleting a structure while it's manipulated. Processes will hold it or contend for it, when they are or want to be

- Creating the cursor
- Alter the cursor

cursor: pin S

The “pin S” wait indicates sleeps on a sharable mutex protection against deleting a structure while it's used. Processes will hold it or contend for it, when they are or want to be in use of a cursor. Nobody wants to have their cursor definition deleted during a SQL statement is still running.

cursor: pin S wait on X

The “pin S wait on X” wait indicates a contention for a shared lock on the pin mutex of a cursor, while it's still held in exclusive mode by another guy. This usually happens when one session tries to execute the cursor, but another one is actually altering it. An example is concurrent DDL on an object used by the cursor we want to reuse.

Issues, Quick Fixes and Long-Term Solutions

Several issues are known regarding cursor-related mutex waits. This paper will elaborate only the most common ones, including some unnecessary Oracle home-made incidents.

Simple: cursor: pin S

Problem

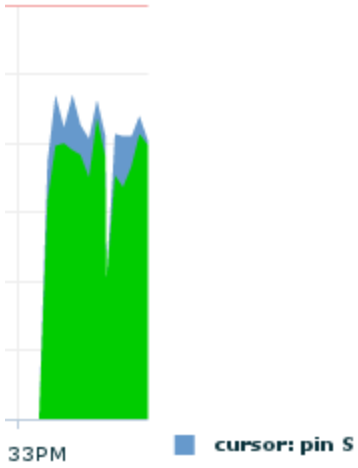

Figure 9: Cursor Pin S Wait

What we see here, is coming from Oracle Enterprise Manager / Top Activity / SQL detail – it's a SELECT. The base (green) field is CPU time, the top (blue) is cursor: pin S waiting (we know, mutex sleeping). The red line above the graph is the number of CPUs, 12 of them in this case.

A part of the green is our mutex spinning, but of course there's other work to so as well (buffer gets, for example). Please note the relation: the cursor: pin S wait most probably won't be a problem here, but nevertheless, it *happens*.

Test Case

This one is the first test case: It's a SELECT, executed within a tight PL/SQL loop. Twenty of those loops are running concurrently, so we are (at least) soft parsing the same cursor for hundreds of times per second.

This is package LOAD7, used for loop in the example:

```
create or replace
PROCEDURE LOAD7(LOOPS IN NUMBER DEFAULT 1000)
is
begin
  for i in 1..LOOPS
  loop
    execute immediate 'select 1 from dual where 1=2';
  end loop;
end;
/
SHOW ERRORS;
/

And that's the corresponding bash script, used to start 20 threads with 10,000,000
executions each:
#!/bin/bash
LP=10000000
sqlplus -s klm/klm <<<"exec load7($LP);"&
sqlplus -s klm/klm <<<"exec load7($LP);"&
sqlplus -s klm/klm <<<"exec load7($LP);"&
sqlplus -s klm/klm <<<"exec load7($LP);"&
sqlplus -s klm/klm <<<"exec load7($LP);"&
sqlplus -s klm/klm <<<"exec load7($LP);"&
sqlplus -s klm/klm <<<"exec load7($LP);"&
sqlplus -s klm/klm <<<"exec load7($LP);"&
sqlplus -s klm/klm <<<"exec load7($LP);"&
sqlplus -s klm/klm <<<"exec load7($LP);"&
sqlplus -s klm/klm <<<"exec load7($LP);"&
sqlplus -s klm/klm <<<"exec load7($LP);"&
sqlplus -s klm/klm <<<"exec load7($LP);"&
sqlplus -s klm/klm <<<"exec load7($LP);"&
sqlplus -s klm/klm <<<"exec load7($LP);"&
sqlplus -s klm/klm <<<"exec load7($LP);"&
sqlplus -s klm/klm <<<"exec load7($LP);"&
sqlplus -s klm/klm <<<"exec load7($LP);"&
```

Result

Cursor: pin S, as demonstrated here, is an indicator for a kind of hot spot object in the Library Cache.

Diagnosis

The most convenient way to find out if and how much we are waiting for cursor: pin S, simply is the wait interface. No matter if one prefers the “Top Activity” screen in Enterprise Manager, AWR- or Statspack reports.

Solution

The basic solution sounds trivial: Don't parse this statement so often. One may laugh, but usually executing a statement that often is not necessary or even a plain bug in the application. In rare cases, usually somehow connected to frequently viewed web pages or parallel batch runs speeding as fast as they can, issuing the same SQL quickly may really be a requirement. In this case, the only solution against too many cursor: pin S waits is to diversify the SQL hash value / SQL_ID of your statement, for example by adding a process- or server name to a comment within its text:

```
select /* WebServer4 */ something from table;
```

Please use this little hack wisely – having too many different SQLs, the database will end up with hard parsing all the time. So *please* do not randomize the comment.

Complex: cursor: mutex S/X

Problem 1

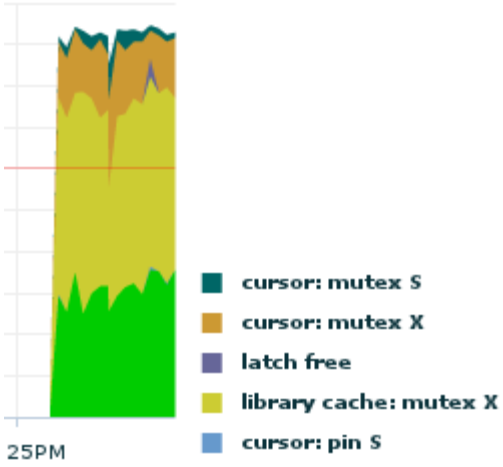

Figure 10: Cursor Mutex S/X Wait

What we have here, is coming from Oracle Enterprise Manager / Top Activity / SQL detail – we see the footprints of an UPDATE. From bottom to top: The green field is CPU time, the second (yellow) a generic library cache exclusive serialization, the third (orange) is our cursor: mutex X wait, and on top in dark petrol, we see a little bit of cursor: mutex S. The red line in upper third is the number of CPUs, 12 of them in this case.

Again, a part of the CPU time is mutex spinning, and still, there's other work to so as well (working on buffers, etc.). We see a massive concurrency situation here, many processes are contending for exclusive usage of cursors. It means, they frequently try to change them, and this is something that deserves closer investigation by the troubleshooter. As the picture shows, the system is heavily overloaded: 12 CPUs, but 20 sessions are actively waiting for something. Users are complaining.

Test Case

The second test case basically is an UPDATE, that's manipulating ten columns of a row, with as many data type changes/variants as possible. The inner loop of package LOAD6 makes sure, that we are going through all variants, coming from a bit structure. If the bit representing the current column is “1”, we are setting it with the NUMBER data type, and if it's “0”, the column will be set with VARCHAR2.

```
/* Code idea and working prototype by Dietmar Bär at ATU, thank you very much! */
create or replace
PROCEDURE LOAD6(p_ChildLimit IN NUMBER DEFAULT 8,
                LOOPS IN NUMBER DEFAULT 1000,
                p_RowNumber IN NUMBER DEFAULT 1)
IS
   TYPE t_arrayc IS VARRAY (11) OF VARCHAR2 (2048);
   TYPE t_arrayn IS VARRAY (11) OF NUMBER;
   TYPE t_bindstrings IS VARRAY (11) OF VARCHAR2 (256);
   v_bindsc        t_arrayc
                      := t_arrayc ('1',
                                   '1',
                                   '1',
                                   '1',
                                   '1',
                                   '1',
                                   '1',
                                   '1',
                                   '1',
                                   '1',
                                   '1');

   v_bindsn        t_arrayn
                      := t_arrayn (1,
                                   1,
                                   1,
                                   1,
                                   1,
                                   1,
                                   1,
                                   1,
                                   1,
                                   1);

   v_BindStrings   t_BindStrings
                      := t_BindStrings (':ZERO',
                                        ':ONE',
                                        ':TWO',
                                        ':THREE',
                                        ':FOUR',
                                        ':FIVE',
                                        ':SIX',
                                        ':SEVEN',
                                        ':EIGHT',
                                        ':NINE',
                                        ':TEN');

   n               NUMBER;
   c               NUMBER;
   d               NUMBER;
   v_BinStr        VARCHAR2 (15);
   v_BitStr        VARCHAR2 (15);
```

```
BEGIN
   dbms_output.enable;
   c := DBMS_SQL.open_cursor;
   DBMS_SQL.parse (
      c,
      'UPDATE /*+LOAD6-1*/ TEN_NUMBERS SET ONE = :ONE, TWO = :TWO, THREE = :THREE,
                                  FOUR = :FOUR, FIVE = :FIVE, SIX = :SIX, SEVEN = :SEVEN,
EIGHT = :EIGHT,
                                  NINE  = :NINE, TEN = :TEN WHERE  ZERO  = :ZERO',
      DBMS_SQL.native);

   DBMS_SQL.bind_variable (c, ':ZERO', p_RowNumber);

   FOR i IN 1 .. LOOPS
   LOOP
      v_BinStr := TRIM (TO_CHAR (dbms_numsystem.dec2bin

                (round(dbms_random.value(1,p_ChildLimit),0)), '00000000000'));

      FOR i IN 2 .. 11
      LOOP
         v_BitStr := SUBSTR (v_BinStr, i, 1);

         IF (v_BitStr = '1')
         THEN
            DBMS_SQL.bind_variable (c, v_BindStrings (i), v_bindsn (i));
         ELSE
            DBMS_SQL.bind_variable (c, v_BindStrings (i), v_bindsc (i));
         END IF;

      END LOOP;

      n := DBMS_SQL.execute (c);
   END LOOP;

   DBMS_SQL.close_Cursor (c);
   COMMIT;
END;
/
/* Code idea and working prototype by Dietmar Bär at ATU, thank you very much! */
```

And that's the corresponding bash script, used to start 20 threads with 64 variants ($CL), 400,000 executions each ($LP), as well as making sure not to hit the same row/block to avoid buffer busy waits (third parameter).

```
#!/bin/bash
CL=64
LP=400000
sqlplus -s klm/klm <<<"exec load6($CL,$LP,1000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,2000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,3000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,4000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,5000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,6000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,7000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,8000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,9000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,10000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,11000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,12000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,13000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,14000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,15000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,16000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,17000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,18000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,19000);" &
sqlplus -s klm/klm <<<"exec load6($CL,$LP,20000);" &
```

Result

As a result of the both scripts combined, we are continuously invalidating cursors by reason BIND_MISMATCH, and generating lots of child cursors. Looking for the matching one, change it anyway and insert a new child cursor all the time, is extremely expensive. In combination with the rapid loop, here 64 variants have been enough to saturate a 12 core machine completely.

Real-life example

Of course, only a DBA going astray will to the above on purpose. But usually for good reason, so this case is a reproduction of a real-life issue out of the Java world.

- The application connects with an Oracle jdbc driver, and the second bind variable (#2) in the cursor is manipulated by a setter method: setNUMBER(2). Obviously, the result is the bind data type NUMBER.
- Next time, we need to set bind #2 to NULL, so the way to go with jdbc is to manipulate the cursor with method setNULL(2). But now the bind variable is of data type VARCHAR2, which leads to cursor invalidation by BIND_MISMATCH, and to creation of a new child cursor.

One or a few occurrences won't harm anybody, but alternately setting around 30 columns to numeric or null this way leads to theoretically 2^30 child cursors for this UPDATE SQL. The effects on Library Cache and its serialization structures are devastating.

In practice, as soon as more than 100 child cursors have been in the Library Cache, the cursor: mutex S/X waits became more and more visible. The server's limitations dictated an upper limit of around 500 cursors, beyond which saturation prevented further activity on the machine.

The system in question was an 8 CPU AIX server, parsing around 100 of those hash-identical UPDATEs per second.

Diagnosis

Initially, this issue may be discovered by measuring the high CPU load or the bad response time. Usually as a next step, DBA personnel will check the Wait Interface with the tool of their choice, and so discovers the mutex wait events discussed.

Finding out why cursors are invalidated are done by a simple query on v$sql_shared_cursor, but be careful: Sometimes the information provided is not very reliable (see pitfalls section later on), a clear recommendation of this paper is to use the quite new 11.2.0.2 event “cursortrace” level 16 following My Oracle Support document [ID 296377.1] to generate a vast, but quite clearly readable trace file. Just search for your SQL_ID in there. Also, if an official Service Request is opened on the case, MOS engineers will ask for this cursortrace dump file immediately.

Having determined the reason for the cursor invalidation, the next challenge is to find out which bind variables are the culprits. Sad, but true: The application developers will need that piece of information to be able to find the position in their code. But good old 10046 level 12 trace dumps out the information we need. Ok, tracing all sessions in such a load situation will be annoying, but

with 11g Oracle introduced another quite cool trace event, named “sql_trace” for a particular SQL_ID, making it easier:

```
alter system set events 'sql_trace [sql:<SQL_ID>]';
```

Lots of trace files will be generated, but you will be able to compare same bind positions with same bind positions. As you can see in the trace excerpt taken during the real-world case from last section above, the ***oacdty*** is different – on this bind position (#2), the data type has changed.

Trace 1:

```
Bind#2
>> oacdty=01 mxl=32(04) mxlc=00 mal=00 scl=00 pre=00
oacflg=03 fl2=1000010 frm=01 csi=873 siz=0 off=168
kxsbbbfp=1118e1cd8 bln=32 avl=00 flg=01
```

Trace 2:

```
Bind#2
>> oacdty=02 mxl=22(22) mxlc=00 mal=00 scl=00 pre=00
oacflg=03 fl2=1000000 frm=01 csi=873 siz=0 off=168
kxsbbbfp=110977db8 bln=22 avl=02 flg=01
value=99
```

Quick Fix

Emergency often makes desperate measures looking good. One such intervention for an issue with too many cursors in the system might be flushing them out frequently. I won't recommend to flush the shared pool, since this will very likely cause other issues. But since version 10.2.0.4+patch or at least with 10.2.0.5 we have the option to purge out one particular parent cursor by using the dbms_shared_pool.purge function.

This is a statement to create a valid dbms_shared_pool.purge command for a known SQL_ID, because we need the address and hash_value of the cursor:

```
select 'exec sys.dbms_shared_pool.purge ('''||address||','||hash_value||''', ''C'');'
               from v$sqlarea
               where sql_id='4dwguxwc8h6gx';
```

In need of purging frequently, suggestion is to create a dbms_scheduler job for kicking out the parent cursor on an appropriate interval (one minute worked great in our real-life example we are talking about). Nevertheless, the database may switch to other problems now,
cursor: pin S wait on X or high hard parsing rates may be side effects of this workaround.

Long-term Solution

Of course, we need a solution for the basic problem: The setter method setNULL(2) we used to manipulate our cursor, shifted the data type of a bind variable formerly being NUMERIC to now being VARCHAR2. The jdbc specification tells us, that handing over the data type to setNULL() is optional. It defaults to VARCHAR2, if we don't. The solution for bind variable #2 to be and to stay on NUMERIC/Integer:

```
setNULL(2, java.sql.Types.INTEGER)
```

This most probably will need to involve development and cause software changes, but it will be a future-proof fix.

Problem 2 – a Similar Case with BIND_LENGTH_UPGRADEABLE

Quite similar to the above case with numeric binds, another problem is known with variant CHAR bind variables.

Technical background: The database does adjust the length of CHAR bind buffers to preset sizes. There are four steps: 32, 128, 2000 and 4000 bytes. So if we are executing a statement with a bind value of 10 bytes, the buffer will be 32 bytes. Do we re-execute it with 52 bytes in the bind variable, the previously created child cursor cannot be reused and will be recreated with a bind buffer of 128 bytes. The system view v$sql_shared_cursor indicates this invalidated child cursor as BIND_LENGTH_UPGRADEABLE.

Having a few of those invalidated children won't harm anybody. They will age out regularly as every invalidated cursor does, but if we have high load and/or concurrency, we may create many of them. How is that?

Think back to the numeric example above: Combinations matter – the more char bind variables we have in a single statement, the more combinations are possible for one cursor. Due to having four possible definitions per bind buffer, the function is y=4^n here, when n is the number of different combinations.

The effect is the similar to what's described in the numeric case: As a result, we are continuously invalidating cursors by reason BIND_LENGTH_UPGRADEABLE, and generating lots of child cursors. Looking for the matching one, change it anyway and insert a new child cursor all the time, is extremely expensive.

Quick Fix

As a quick and dirty solution, the emergency measure using the dbms_shared_pool.purge function as described above, works here as well. Keep in mind, that trading in parsing CPU load in exchange for Mutex Waits may be a bad idea.

Long-term Solution

This is the complicated part, because here is nothing that can be said in general. Your business case and application pro-actively has to avoid a constellation problem.

Heavy: Oracle internal pitfalls

Behavior change from 10g to 11g

Oracle 10g per definition was also vulnerable for cursor mutex issues, because it already had similar child cursor structures, mutexes, cursor invalidation mechanisms. But the changes (improvements!) around Cardinality Feedback and Adaptive Cursor Sharing seem to have made it more delicate for parsing issues. If we have a look at the Harder Soft Parse concept described above, that's somehow understandable. Oracle followed several approaches to improve the speed of cursor handling. One of them – to slim the kernel code – works to the disadvantage of applications producing too many child cursors.

The question was, why a system on 10g did not produce tons of child cursors, while the new 11g database does. Here 's the official answer from Oracle:

"It's important to note that cursor obsoletion code was removed in version 11. That means we no longer obsolete a parent cursor when it reaches 1024 child cursors."

As a workaround, Oracle recently published Enhancement Patch 10187168, that introduces the hidden parameter "_cursor_obsolete_threshold". As usual, only use underscore parameters under guidance of Oracle Support.

Later, this patch became part of 11.2.0.3.0.

Bugs like 10157392 and 12939876

Both bugs are related and based upon each other. They are about a memory leak in 11g code, that causes a creeping increment of child cursor numbers, especially when the Shared Pool is under stress. In the end, we have all cursor: mutex wait events discussed above without doing anything wrong. The usual diagnosis via v$sql_shared_cursor does not show any reason, and cursortrace indicates this bug by dumping lots of (thousands) Bind mismatch(3) cursor versions.

The bugs are fixed in 12.1, and back-ported to 11.2.0.3. No fix is included in any 11.2.0.2 patch bundle.

Bug 9591812

This bug is about wrong wait events in 11.2 ("cursor: mutex S" instead of "cursor: mutex X"). The kind of official MOS workaround says: "Be cautious when interpreting S mode mutex / pin waits." So we are officially told not to believe or at least to suspect anything what we see.

The bug is fixed in 12.1.

Summary

Cursor mutex wait events are a problem deeply related to Oracle internal mechanics, and applications using the transparent possibilities of the Oracle RDBMS. Since we implicitly have to use new features of the parser and other built-in functions, applications have to make sure that they are aligning to specifications. Oracle can't know what third-party development departments all over the world are doing, they just can try to come along with their own specifications when publishing new functionality. That they are not inerrant, all DBAs and database developers know. The bugs described above are a result of developing an incredibly complex product, and we will have to live with such issues. Database administration personnel has to try to stay tuned on news and known issues, as well as staying in exercise with troubleshooting techniques.

For "Resolving Child Cursor Issues Resulting In Mutex Waits", there are some suggestions for casualties of this topic:

- Check how the application does handle its bind variables. Avoid BIND_MISMATCH at any cost.
- Reduce the number of cursor versions (= child cursors) to below 100, every single child will increase load and effort.
- If in trouble, look up MOS for Oracle bugs matching your issue. They will be there.
- Upgrade your 11g to 11.2.0.3 or higher. Especially 11.2.0.2 is the tail-end charlie in cursor handling.

Additional Readings

My Oracle Support (MOS) Document IDs

- 1356828.1
- 1377998.1
- 296377.1

Authors

- Põder, Tanel
  Year 2011 *Presentation: "Oracle Latch and Mutex Contention Troubleshooting"*
  http://www.slideshare.net/tanelp/oracle-latch-and-mutex-contention-troubleshooting
- Shallahamer, Craig
  Book: "Oracle Performance Firefighting" (ISBN 978-0-9841023-0-3)
  http://resources.orapub.com/Oracle_Performance_Firefighting_Book_p/ff_book.htm
- Nikolaev, Andrey
  Year 2011 *Blog entries: "Mutex waits. Part 1 + 2"*
  http://andreynikolaev.wordpress.com/2011/07/09/mutex-waits-part-1-%E2%80%9Ccursor-pin-s%E2%80%9D-in-oracle-10-2-11-1-invisible-and-aggressive/
  and
  http://andreynikolaev.wordpress.com/2011/10/25/mutex-waits-part-ii-cursor-pin-s-in-oracle-11-2-_mutex_wait_scheme0-steps-out-of-shadow/

Company

Klug GmbH integrierte Systeme is a specialist leading in the field of complex intra-logistical solutions. The core competence is the planning and design of automated intra-logistics systems with the main focus on software and system control.

Product

The company's product series, the “Integrated Warehouse Administration & Control System” iWACS® is a standardized concept of modules offering the tailor-made solution for the customer's individual requirements in the field of intra-logistics. With the iWACS® concept, customers do not only gain maximum flexibility, but also an optimal cost/performance ratio.

Developed with state-of-the-art software technologies, the iWACS® module .WM is an adaptable standard software granting flexibility and system stability. The warehouse management system .WM controls and organizes all tasks from goods-in to dispatch and is up to all requirements from manual warehouse to fully automated distribution center. To interlink .WM with any ERP system, customers can dispose of tried and tested standard functions.

Author

Being in IT business as a specialist for Linux, **Martin Klier** has been involved in the administration of Oracle Databases about ten years ago, and works on senior level for more than three years now. The integration of large Oracle-based high-availability solutions (MAA) with RAC and Data Guard have been the first challenges, in the last five years he largely moved into performance analysis and tuning of complex systems with dynamically changing load profiles.

Martin frequently blogs about current issues and their solutions at http://www.usn-it.de.

Address

Martin Klier
Klug GmbH integrierte Systeme
Lindenweg 13
D-92552 Teunz

Telefon:	+49 9671 9216-245
Fax:	+49 9671 9216-6245
E-Mail	martin.klier@klug-is.de
Internet:	http://www.klug-is.de
priv. Weblog:	http://www.usn-it.de