accentureoperations

ORACLE MUTEXES

FRITS HOOGLAND

ABOUT

Principal consultant with the Accenture Enkitec Group.

Oracle database performance, configuration and capacity specialist. **Likes hard problems**!

25 years of experience with the Oracle database.

Hardware, networking and operating system experience. Cloud is a combination of this.

IT infrastructure automation and other databases.

Technical (co-)reviewer of books (Kyte's Database Architecture, others), co-writer (Expert Exadata).

Member of the Oaktable.

Oracle ACE Director.

email: frits.hoogland@accenture.com
twitter: @fritshoogland
blog: https://fritshoogland.wordpress.com

ASSUMPTIONS AND WARNING

This presentation **has nothing to do with daily operations**.

It is research and information about a single, specific mechanism in the Oracle database.

If you do not like technical detailed information, or are searching for something general and applicable to daily operations, this talk is not for you. In fact, it'll probably bore you to death. We can't have that.

This will help you to build a fundamental understanding of the actual working of the database. Which gives you the ability to do troubleshooting, performance tuning, and anything else that requires to understand how the database truly works.

I do assume the attendee to have a reasonable knowledge of general concepts like mutex/spinlocks, how the database works, dynamic performance views, what undocumented parameters are.

VERSION

The research for this presentation is done in Oracle 18.4.

Operating system: Oracle Linux 7.5 (a vagrant box from Bento)
Kernel: 4.1.12-124.18.6.el7uek.x86_64

The inner mechanics of the Oracle database can change with a single patch, without **any** warning or documentation. The general tendency is mechanics do not change that much and that fast.

It does mean that if specific details are important, it makes sense to investigate/validate it after any change to the database. Change here is one-off patch, PSU or version. Anything that potentially changes the executable.

WHAT IS A MUTEX

An **Oracle** database mutex is a memory structure to manage concurrency.
This is what Oracle database latches have been doing for a long time.

An Oracle database mutex is something different than an operating system mutex.
Although both apply the same principle of managing concurrency.

All of these are implementations of spinlocks.
That means that if a process finds the spinlock been taken in an incompatible mode,
it will retry getting it until it succeeds.

The idea is that spinlocks are only briefly held, therefore the spinning should only last a very short time.

HISTORY: UP TO VERSION 11.2.0.3

Mutexes are reported to have been gradually implemented since Oracle 10.2.

Up until approximately version 11.2.0.3, there was no back-off mechanism for mutexes (bug 10411618).
(like there is for latches, which for which the first mechanism is sleeping, the second one is a semaphore wait)

No back-off mechanism could lead to processes spinning for a mutex, totally occupying a CPU.
For systems maxed out on CPU already, this could lead to a death-spiral if spinning for mutexes is added, because if these procs start out, they are newer/have higher priority, pushing the the process it is waiting for off CPU.

Even without mutexes, if your load exceeds your *actual* CPU capacity, it takes away to possibility for userland processes to schedule and execute actions at will, which means the order of things to happen gets random.

The database resource manager should be able to manage excess process usage.

HISTORY: VERSION 11.2.0.4+

Since (base release) 11.2.0.4, mutexes have a back-off mechanism.

The back-off mechanism is implemented using wait scheme's.

The default scheme is 2 (_mutex_wait_scheme):

- a process spins 255 times (_mutex_spin_count).
- and then sleeps 10ms (_mutex_wait_time=1).

This prevents spinning mutexes from consuming a full CPU.

Sleeping is implemented using semaphores, so a process can get woken if there is need.

LATCH ALLOCATION AND VISIBILITY

Latches can be:

- Parent latches, which are located in fixed SGA at a consistent location/address. Pointer to (array of) children.
- Child latches, which are located in the shared pool, number derived from instance parameters.

This means that all times, the number and location of latches is known.
Therefore, there are views like V$LATCH/V$LATCH_CHILDREN/V$LATCH_PARENT
that contain the status of every distinct latch.

A latch could protect multiple independent resources.
An example of that is the cache buffers chains latch.

MUTEX ALLOCATION AND VISIBILITY

A mutex is (dynamically) created inside and as part of the object it protects, such as library cache hash bucket, library cache handle, row cache parent, etc.

Therefore the mutex is also removed if the object is removed.

Probably to make the overhead of a mutex as low as possible,

there is no central registration of individual mutex existence and individual statistics.

This is a major difference with latches.

Mutexes are implemented at specific code locations:

• KKS (kernel compile shared cursors)	child cursor concurrency in parent cursor heap 0.
• KGL (kernel generic librarycache)	library cache hash table/handle concurrency.
• KQR (kernel query rowcache)	rowcache concurrency.
• KDLW (kernel data LOB write)	unknown (not investigated)

accentureoperations

MUTEX VIEWS

There are following views that provide mutex information:

X$MUTEX_SLEEP -> V$MUTEX_SLEEP
Cumulative instance wide mutex sleeps per location, wait time in in microseconds.

X$MUTEX_SLEEP_HISTORY -> V$MUTEX_SLEEP_HISTORY
Instance wide mutex sleeps per occasion, in an array hashed by MUTEX_ADDR and BLOCKING_SESSION.
(not in a circular buffer, as the documentation suggests, via http://andreynikolaev.wordpress.com)

DBA_HIST_MUTEX_SLEEP
AWR snapshot of V$MUTEX_SLEEP.

CDB_HIST_MUTEX_SLEEP
CDB AWR snapshot of V$MUTEX_SLEEP.

AOL / UOL - USER/MUTEX STATE OBJECTS

When a process is going to be accessing a mutex, a helper structure as state object is going to be created upfront.

This is a structure known as UOL (user operation log), and contains AOL's or Atomic Operation Log entries.

A change or the intention for a change to a mutex is written to an Atomic Operation Log entry.

For example: this is an entry of an exclusive get:

```
KGL-UOL SO Cache(total=131, free=107)
KGX Atomic Operation Log 0x7e58fb28
 Mutex 0x61853818(209, 0) idn 1078 oper EXCL(6)
 Library Cache uid 209 efd 5 whr 62 slp 0
 oper=0 pt1=0x7e780b88 pt2=(nil) pt3=(nil)
 pt4=(nil) pt5=(nil) ub4=1399 flg=0x1 uw1=0 uw2=0
 msk=0000-0000-0000-0000-0000
```

And this is an entry of another process trying to obtain the same mutex:

```
KGL-UOL SO Cache(total=128, free=16)
KGX Atomic Operation Log 0x7e4a8d28
 Mutex 0x61853818(209, 0) idn 1078 oper GET_EXCL(5)
 Library Cache uid 403 efd 4 whr 62 slp 5179
 oper=0 pt1=(nil) pt2=(nil) pt3=(nil)
 pt4=(nil) pt5=(nil) ub4=0 flg=0x1 uw1=0 uw2=0
 msk=0000-0000-0000-0000-0000
```

AOL / UOL - USER/MUTEX STATE OBJECTS

An UOL is created using the C function kglGetSessionUOL.

An UOL is allocated in a shared pool area that contains the session’s state, called KKSSP^*SID.*

Inside the KKSSP, the UOL is allocated in an area (subpool) called kglss.

KKSSP probably means kernel compile session state pages.

Multiple UOL's can be allocated at the same time, they are dependent on mutex change occasion and type, they are not a session scoped general pool of AOL's.

An AOL entry in the UOL is allocated using the C function kgxAOLInit.

The function of the AOL is to have log like metadata, for example to make cleaning them up in case of a session crash possible.

The UOL is a state object, and that is one of the functions of a state object.

PMON cleans up orphaned resources if this is necessary.

The KKSSP^*SID* area is also documented to contain the state objects of pins and (library cache) locks.

(bug: 23315153, note: 2151847.1, note: 2369127.1)

MUTEX DUMP

There is no dump or event that I am aware of to dump mutexes to a file.
The reason is the instance does not know where they are.

However, you can dump the UOL's a session has allocated.

A systemstate dump shows all UOL's of all sessions, and thus all mutexes held in that instance.
(oradebug dump systemstate 266)
See MOS note 423153.1, Reading and understanding systemstate dumps.

A processstate dump shows all UOL's of a single process.
(oradebug dump processstate 10)

LIBRARY CACHE: BUCKET MUTEX X

P1: bucket number

```
select kglhdpar PARENT_ADR,
       kglhdadr HANDLE_ADR,
       kglhdnsd NAMESPACE,
       kglobtyd OB_TYPE,
       kglobt03 SQL_ID,
       kglnaobj OB_NAME,
       case when kglobt09=65535 then 'PARENT' else to_char(kglobt09) end CHILD#,
       kglobhd0 HEAP0_ADR,
       kglobhd6 HEAP6_ADR
from   x$kglob
where  kglhdbid = p1;
```

P2: blocking SID (in the 'high order bits', high 32 bits of 64 bits number)

```
select bitand(p2/power(2,32),power(2,16)-1)
from   dual;
```

P3: location_id (in the 'lower order bits, low 32 bits of 64 bits number)

```
select location
from   x$mutex_sleep
where  location_id = bitand(p3,power(2,16)-1)
and    mutex_type = 'Library Cache';
```

LIBRARY CACHE: BUCKET MUTEX X

This is a wait for an exclusive (mutex X) get of the mutex of the specific library cache bucket.

Any time a bucket is read to find a library cache handle, this mutex is taken exclusively.

The reasons for the exclusive get is twofold:

1. Being able to uniquely raise the mutex counter of this bucket by one.

   I can only think of excessive external factors like CPU oversubscription or heavy swapping for this to cause this to take a long time.

2. In case of parent handle creation, having unique access to the bucket chain to add the parent handle to it.

   If the parent handle must be created, after obtaining the bucket mutex, the function to obtain the parent handle (kglhdgn) calls kglhdal to allocate the handle. This function takes the shared pool latch to get unique access to the freelist to allocate memory. After the memory is allocated, the parent handle mutex is created and taken linked in the handle pointer chain, after which the bucket mutex is released.

   So shortage of shared pool memory can cause longer wait times for this event.

LIBRARY CACHE: MUTEX X

P1: library cache hash value

```
select kglhdpar HANDLE_ADR,
       kglhdnsd NAMESPACE,
       kglobtyd OB_TYPE,
       kglobt03 SQL_ID,
       kglnaobj OB_NAME,
       kglobhd0 HEAP0_ADR
from   x$kglob
where  kglnahsh = p1
and    kglhdpar=kglhdadr;
```

P2: blocking SID (in the 'high order bits', high 32 bits of 64 bits number)

```
select bitand(p2/power(2,32),power(2,16)-1)
from   dual;
```

P3: location_id (in the 'lower order bits, low 32 bits of 64 bits number)

```
select location
from   x$mutex_sleep
where  location_id = bitand(p3,power(2,16)-1)
and    mutex_type = 'Library Cache';
```

LIBRARY CACHE: MUTEX X

This is a wait for an exclusive (mutex X) get of the mutex of the specific library cache (parent) handle.

Any time any data in the parent handle or child handles needs changing, like for a library cache lock, library cache pin, library cache load lock, any statistics (invalidation count, execution count, load count, active locks, broken count, etc.) this mutex is taken.

The reasons for the exclusive get are twofold:

1. Being able to uniquely raise the mutex counter of this parent handle by one.

   I can only think of excessive external factors like CPU oversubscription or heavy swapping for this to cause this to take a long time.

2. In case of pinning, the heaps that are needed are allocated while holding this mutex.

   If the heaps need allocating, typically during pinning, it's done while holding this mutex. The function that does so is kglobal. In this function the shared pool latch is taken to allocate memory. It might need to free chunks by reading their function, and deallocate them executing the appropriate code to unregister these.

   So shortage of shared pool memory can cause longer wait times for this event.

LIBRARY CACHE: DEPENDENCY MUTEX X

I haven't found a way to produce this wait event in a reasonable real life scenario.

What I see happening is the handle mutex is taken prior to taking the dependency mutex, so any wait occurs for the handle when I try to artificially cause a wait for the dependency mutex.

This is what the data dictionary says:

```
SQL> select name, parameter1, parameter2, parameter3 from v$event_name where
lower(name) like '%mutex%';

NAME                                     PARAMETER1 PARAMETER2           PARAMETER3
---------------------------------------- ---------- -------------------- ----------
SecureFile mutex
row cache mutex                          cache id   where requested
cursor: mutex X                          idn        value                where
cursor: mutex S                          idn        value                where
library cache: mutex X                   idn        value                where
library cache: bucket mutex X            idn        value                where
library cache: dependency mutex X        idn        value                where
library cache: mutex S                   idn        value                where
```

In other words: the parameter description is standard, so I don't know what p1/2/3 truly are.

LIBRARY CACHE: DEPENDENCY MUTEX X

The dependency tables are allocated outside of the mutex get, before the mutex get for the handle.

Logically reasoning, a separate mutex for changing the handle dependency table solves excessive holding the handle mutex if the number of dependencies is high. Think SYS.DUAL.

But I can't come up with a scenario where this truly would block, and thus where the time lies.

Because the dependency table memory is allocated outside of holding the mutex, the most logical cause for contention is concurrency.

CURSOR: *

P1: library cache hash value

```
select kglhdpar HANDLE_ADR,
       kglhdnsd NAMESPACE,
       kglobtyd OB_TYPE,
       kglobt03 SQL_ID,
       kglnaobj OB_NAME,
       kglobhd0 HEAP0_ADR
from   x$kglob
where  kglnahsh = p1
and    kglhdpar=kglhdadr;
```

P2: reference count (in the 'lower order bits', low 32 bits), blocking SID (in the 'high order bits', high 32 bits)

```
select bitand(p2,power(2,16)-1)             REF_COUNT,
       bitand(p2/power(2,32),power(2,16)-1) BLOCKING_SID
from   dual;
```

P3: location_id (in the 'high order bits, high 32 bits of 64 bits number)

```
select location
from   x$mutex_sleep
where  location_id = bitand(p3/power(2,32),power(2,16)-1)
and    mutex_type in ('Cursor Parent','Cursor Pin','hash table');
```

CURSOR: *

The mutexes on the basis of the cursor: * wait events are allocated in heap 0 of the parent cursor.

There are several mutexes here:

1. Cursor parent. This is the main mutex for access to the parent heap 0.
2. Hash table. This is the list of pointers to children in the parent heap 0.
3. Cursor pin. This is the pin status of the parent heap 0.

The most common wait to encounter is: cursor: pin S wait on X

Which means a session needs to find a suitable child from the child list. That is done by scanning the child list from the most recent child to the oldest (so in reverse of creation order), and needs to pin each child in S mode for investigation. This wait means the session tries to pin a child in S mode, but it can't because it's currently pinned in X mode. This typically happens when a child is being created.

If you look at the location_id (p3), you'll find: cursor pin, kkslce [KKSCHLPIN2] ; kks lock child entry.

ROW CACHE MUTEX

P1: row cache parent cache id

```
select distinct kqrstcid P_CACHE_ID,
       kqrsttxt        NAME
from   x$kqrst
where  kqrstcid = p1;
```

P2: location_id

```
select location
from   x$mutex_sleep
where  location_id = p2
and    mutex_type = 'Row Cache';
```

P3: not used

ROW CACHE MUTEX

This is a wait for any row cache mutex.

Any means the bucket mutex for the cache, or parent object mutex.

If a row cache parent object does not exist, it needs to be allocated before the row can be read into the cache.

The allocation of the parent cache memory is done while holding the row cache parent object mutex.

The allocation comes from the shared pool, and requires the shared pool latch for getting memory. So shared pool shortage can influence the time the mutex is held.

After the mutex of the parent object is taken, the parent enqueue is set, and the mutex is freed.

If the row cache enqueue is set, any session trying to read that parent in an incompatible mode will encounter a 'row cache lock' wait event. (p1=cache id,p2=mode, p3=requested; 0=null, not locked, 3=shared mode, 5=exclusive)

If a row cache entry needs to be read, it is done holding the row cache lock.

If a row cache entry is not present, it's read recursively, holding the row cache lock.

Some row cache gathering information is shown with event 10222.

MUTEX OPCODES

```
(gdb) set $op=&kgxOpcodeName
(gdb) x/s *$op++
0x128e3f18:     “NONE”              0
0x1475b514:     “GET_SHRD”          1
0x146b5b80:     “SHRD”              2
0x1475b520:     “SHRD_EXAM”         3
0x1475b52c:     “REL_SHRD”          4
0x1475b538:     “GET_EXCL”          5
0x12d3f754:     “EXCL”              6
0x1475b544:     “REL_EXCL”          7
0x1475b550:     “GET_INCR”          8
0x1475b55c:     “INCR_EXAM”         9
0x1475b568:     “GET_DECR”          10
0x1475b574:     “DECR_EXAM”         11
0x1475b580:     “RELEASED”          12
0x1475b58c:     “EXCL_SHRD"         13
0x1475b598:     “GET_EXAM”          14
0x1475b5a4:     “EXAM”              15
0x1475b5ac:     “GET_LONG_EXCL”     16
0x1475b5bc:     “REL_LONG_EXCL”     17
0x1475b5c0:     “LONG_EXCL”         18
```

accenture operations

MUTEX INTERNAL C FUNCTIONS

The following functions are used by Oracle to manipulate mutexes (not a complete list!):

kglGetSessionUOL(*generic PGA,?) - Initialize AOL entry in KKSSP^SID, returns pointer to AOL.

kglGetBucketMutex(*generic PGA,?,?) - Get mutex for KGL hash table bucket, calls kglGetMutex.

kglGetMutex(*generic PGA,*mutex,**AOL,?,?,*?) - General KGL layer call for getting a mutex.
Get mutex, includes different calls and validations.

kgxExclusive(*generic PGA, *mutex, **AOL) - Get mutex in EXCL mode (6).

kglMutexHeld(*generic PGA, *mutex, **AOL) - Validate mutex state with AOL.

kglReleaseBucketMutex(*generic PGA,?,?) - Release mutex for KGL hash table bucket, calls kglReleaseMutex.

kglReleaseMutex(*generic PGA, *mutex, **AOL) - General KGL layer call for releasing a mutex.
Release mutex, includes different calls and validations.

kglIsMutexHeld(*generic PGA, *mutex) - Validate mutex, returns pointer to AOL.

kgxRelease(*generic PGA, **AOL) - Set mutex in NONE mode (0).

kgxShared(*generic PGA, *mutex, **AOL) - Get mutex in SHRD mode (2).

kgxAOLInit(*generic PGA,?,?) - Initialize AOL entry for functioning as lock?

kgxLongExclusive(*generic PGA, *mutex, **AOL) - Get mutex in EXCL mode (6), only used in kxsGetRuntimeLock
to pin cursor in exclusive mode.

MUTEX INTERNAL C FUNCTIONS

kgxExclusiveNoWait(*generic PGA, *mutex, **AOL) - Get mutex in EXCL mode (6), used for immediate deletion.

kgxExclusive2Shared(*generic PGA,*?) - Change mutex from EXCL (6) to SHRD (2) mode.
Only used in kxsDowngradeLock to change pin state of cursor.

kgxDecrementExamine(*generic PGA, *mutex, **AOL) - Decrement RefCnt Examine.

kgxEndExamine(*generic PGA,?,*?) - Clear examine mode

kgxSharedExamine(*generic PGA, *mutex, **AOL) - Set shared examine mode (SHRD_EXAM, 3)

kgxExamine(*generic PGA,?,*?) - Set examine mode (EXAM, 15)

kgxIncrementExamine(*generic PGA,?,*?) - Increment RefCnt Examine.

kgxModifyRefCount(*generic PGA,**AOL,?,?) - Increase or decrease reference count for a mutex in examine mode.

kgxWait(*generic PGA,**?,?,?,*?,?) - Register wait and wait using post-wait.

kglMutexNotHeld(*generic PGA,*mutex) - ??

SO...HOW MUCH ARE MUTEXES REALLY BEING USED?

It depends...

But quite a lot.

If time permits: demo!

CONCLUSION

In recent versions, I have not encountered excessive mutex issues.

Under normal circumstances, mutex waits means **concurrency**.

Mutex issues are more likely to occur when the environment forces non-sequential execution.
This is fancy speak for CPU oversubscription forcing random sleeps on processes.
A simple formula to determine if your machine is oversubscribed:
(total) AAS > # CPU cores

(so 100% CPU usage is not optimal use of machine and CPU)

In most mutex issues that I encountered, it eventually turned out to be CPU oversubscription amplifying mutex waits.

Questions and Answers

Thank you for attending!

accentureoperations

gdb script used to show mutex handling:

```
set pagination off
rbreak ^kgx[A-Z]
commands 1-26
silent
print $rip
c
end
```

(change 26 to the number of functions found with the ^kgx[A-Z] regular expression)