Conquering Oracle Latch Contention

(Identifying, measuring, and resolving harmful latch contention)

Latest version available at http://www.orapub.com

This page has been intentionally left blank.

Table of Contents

Conquering Oracle Latch Contention

Craig A. Shallahamer (craig@orapub.com)

Original June 7, 2004

Version 1d June 23, 2004

Abstract

As Oracle server complexity, transaction throughput, and simultaneous usage all continue to increase, latch contention can plague even the most experienced Oracle performance specialist. This paper describes why latches exist, how they work, how to detect latching contention, and how to resolve the contention. To demonstrate this process, the library cache latch will be used throughout the paper as well as publicly available latching scripts.

Introduction

It's really true. Even the most seasoned Oracle performance specialists shudder at the thought of dealing with Oracle latch contention. I have heard some of the most respected Oracle performance specialists say that once you get latch contention, there really isn't a whole lot you can do. This is absolutely wrong! Why do even the best performance specialist's knees buckle in the face of latch contention? Because it means understanding a great deal about not only latching, specific Oracle architecture internals, and some queuing theory [PPM], but it also means taking risks like suggesting bold application changes or implementing hidden instance parameters that few have actually tried in a real production environment.

Over my years as a consultant and a teacher, I have found the best way to not only resolve but to teach others how to resolve latch contention is to first understand Oracle's general latching algorithm and then understand the specific Oracle architecture component that the latch is related to. So it's a two-step educational process that breaks out into a seven-step contention resolution strategy. In this paper, I'll discuss the general latching algorithm and a supporting seven-step process to identify, measure, and resolve latch contention. However, I will not discuss the Oracle architecture internals related to each latch. There are many resources available, including my *Advanced Reactive Performance Management* [RPM] class.

Unless specifically mentioned, all the tools and scripts mentioned and used in this paper are available for free on OraPub's web site, www.orapub.com. The tools are combined into a single toolkit called the *OraPub System Monitor* or OSM for short. [OSM]

How To Lean About Oracle Latching

As I mentioned in the *Introduction*, I have found the best way to not only resolve but to teach others how to resolve latch contention is to first learn about Oracle's general latching algorithm and then learn about the specific Oracle architecture area that is related to the latch. It's kind of like, once you learn the principle then you can apply it to the specific problem. Like honesty, once you learn to live honestly, actually being honest in various situations just naturally occurs and makes sense. Latching, in some ways, is just like this.

Latches protect Oracle memory structures. And while there are many Oracle memory structures, and therefore many Oracle latches, they all operate under the same basic algorithm. So the first step is learning about this algorithm. The second step is to learn about the memory structures. Once you understand both the latching algorithm and the memory structure architecture, then the solutions naturally come forth in those, all to infrequent, "ah ha" moments.

The Process Explained

There are seven steps to detecting, measuring, and resolving latch contention. The steps are summarized below. Throughout the paper, I will explain each of the steps in more detail complete with actual examples.

1. **Understand the general latching algorithm**. I've already mentioned this above and I'll detail the algorithm in the next section.

2. **Detect latch contention**. While it may be obvious to you, before you attempt to resolve latch contention, make sure resolving latch contention is worth your time and will significantly improve performance. The best way to detect latch contention is using a response time based approach with its core based upon Oracle's wait event interface. [RTA, SWA, RPM] For example, if response time is unacceptable and an Oracle process or processes are waiting 90% of the time for a latch, then it makes sense to spend your time resolving the latch contention. But if that percentage is only 15%, then you would had better focus elsewhere. While you are celebrating your 15% response time improvement, the users will be planning your demotion because of the other 85%. I will discuss this in more detail later in the paper. (Not your demotion, but detecting latch contention.)

3. **Determine the latch**. Once you know there is significant and harmful latch contention, you will need to determine which specific latch. This may seem obvious, but depending on your version of Oracle and if you are looking at performance interactively (i.e., in real time) or historically, you may only know there is latch contention, but not the specific latch.

4. **Understand the related kernel code**. This is when it becomes important to understand what the latch is actually protecting and why. For example, if you have library cache latch contention, then you will need to know what the library cache is, how it works, and what you can do to affect how it works. Understanding latching in general is good, but if you don't understand the underlying architecture, your attempts to resolve the contention will be nothing more than a good guess. As a side note, you will also be able to quickly understand why others' recommendations could never solve the latch contention.

5. **Understand the nature of the latch contention**. Ask yourself two questions and how you could affect the answer to the questions. The first question is, "Why is the latch held so long?" And the second question is, "Why is the latch being requested so often?" These are two very different questions that address two distinct yet closely related operations or the nature of how the CPU subsystem and Oracle's latches are working together.

6. **Devise multiple resolution strategies**. Because of uptime requirements, response time requirements, politics, and the list goes on and on, you will need to come up with multiple ways to possibly resolve the latch contention. Hopefully one of your ideas will be able to be implemented. Many times, latching contention solutions require unusual changes in the system that can not or will not be allowed in your IS group. So you want to have as many options and fallback plans as possible.

7. **Take appropriate action to resolve**. Finally…you have methodically worked through this process and are ready to actually implement a change that will hopefully reduce the latch contention. If you have followed these seven steps, you stand a very, very good chance of improving response time.

Understanding The General Latching Algorithm

The first step to ultimately resolve latch contention is to understand Oracle's general latching algorithm. But even before that, it's important to understand how Oracle uses locks and why it uses locks, and then understanding what an Oracle latch is and why it is used. Then finally, we are in a position to understand Oracle's latching algorithm and more advanced topics like how multiple latches are implemented. This section very quickly discusses each of these topics.

Types of Oracle Locks

While you never read or hear this in official Oracle Corporation presentations, I feel that Oracle has three basic ways of protecting things. There are application locks, data dictionary locks, and memory structure control.

Application Locks are under the control of an application developer. For example, if I type, "lock table employee exclusive" and you type, "update employee set salary = salary * 0.75" (which is not a nice thing to do by the way), then thankfully your operation will be blocked…locked. Any DBA can observe this by looking at the v$lock view and also the wait event views with an event name of "enqueue wait" of type TX (row or block related) or TM (table related).

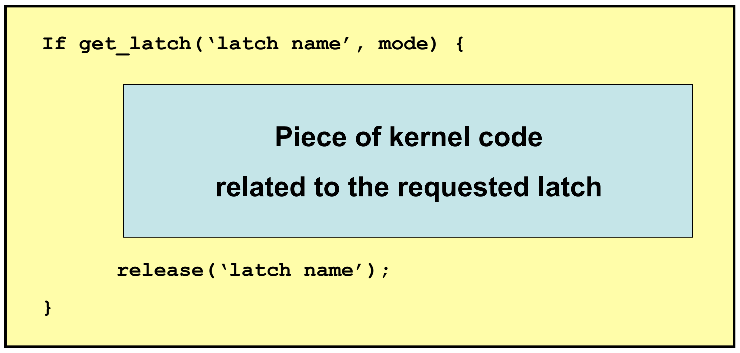
Data Dictionary Locks are under the control of Oracle Corporation kernel code developers. Keep in mind, that when you type something like "create table…." Oracle must insert a row into sys.tab$, sys.col$, and others. These tables must be appropriately locked just like the Application Developer does. We don't have control over these locks because we didn't write the Oracle kernel code. Just as with application locks, any DBA can observe data dictionary locks by looking at the v$lock view and also the wait event views with an event name of "enqueue wait" of type TX (row or block related) or TM (table related).

Memory Structure Control is under the control of Oracle Corporation kernel code architects and developers. Unlike structures that are related to tables and indexes, memory structure control is related to the memory-based structure that resides in Oracle's cache. So while they are not "locks" in the pure sense, they are protecting something…and that something is memory. We call these protection structures Oracle *latches*. Unlike enqueues, Oracle latches do not maintain any order. As it will be apparent later, the first process to ask for a latch could probabilistically be the last person to receive the latch. (Thankfully it is an extremely small probability. So small, I've never seen this occur.)

The Oracle Latch

Two definitions that have stood the test of time are: Oracle latches ensure serial execution of Oracle kernel code and Oracle latches ensure Oracle's cache is not corrupted resulting in physical data corruption (which can lead to an ugly lawsuit). I like both definitions because each highlights a significant aspect of an Oracle latch.

To ensure the memory structure is not corrupted, certain Oracle memory structures can only be accessed by one and only one process. Even when reading a memory structure, Oracle must ensure it is not changed by another process. Oracle has chosen to use a simple method to protect memory structures and it's called a latch. Latches protect a memory structure by surrounding any kernel code that will access the memory structure with a latch. The latch that surrounds the library cache kernel code is called the *library cache latch*. Before the kernel code can be run, the latch must be acquired (a process I'll describe below) and when a CPU is done running the specific piece of kernel code the latch is released.



```
If get_latch('latch name', mode) {

      release('latch name');
}
```

Figure 1. Latches Surround Kernel Code. Oracle protects memory structures by ensuring that any kernel code that accesses the structure must first get the appropriate latch. For example, if the kernel code is about checking to see if a buffer resides in the cache, the cache buffer chain kernel code will need to be executed. But before this can happen, one of the appropriate cache buffer chain latches must be acquired.

A latch can be requested in two different ways. A process (i.e., server or background) could ask for a latch one time and if it does not get the latch, control is returned to the process (which could ask for different latch or something entirely different). This is called asking for a latch in *immediate* mode. A process could also ask for a latch once, but if the latch is not received immediately, it keeps trying (a process I'll describe below) until the latch is acquired. This is called asking for a latch in *willing to wait* mode. Oracle Corporation architects decide which mode is appropriate.

The General Oracle Latching Algorithm

The general Oracle latching algorithm is actually quite simple…and it should be because it must be fast, very fast. Protecting memory structures is something you don't want to spend a lot of time on. (Just how much time? That's why we have response time analysis and the wait event views. [SWA, RTA])

In a nutshell, when a process asks for a latch in *immediate mode*, if the latch is not available from just one request (sometimes called a *fast get*), control is returned to the process. When a process asks for a latch in *willing to wait mode*, the process asks for the latch once (just like in *immediate mode*), but if the latch is not acquired the "spin and sleep" latching algorithm kicks in. Basically, the *spin and sleep* algorithm process repeatedly asks for the latch and if the process does not get the latch it takes a break and sleeps.

When a process repeatedly asks for a latch, called *spinning on the latch*, it's like a child asking for candy over and over (but really fast). It would sound like "gimme, gimme,…". This continues until either the latch is received or until the "gimme…" occurs *spin_count* times. If the spinning does not work, that is, the process does not get the latch, the process will drop into the sleep cycle and sleep for a specified number of milliseconds. (In general, the specified wait time…I mean sleep time increases exponentially.) When the process wakes up, it spins once again. If the latch is not acquired once again, then it sleeps once again. This cycle continues until either the process gets the latch or the user gets so stink'n frustrated, they break out of the "lock."

Let's talk about timing for a second because this is really interesting to those of you who like to keep track of time. Response time is composed of service time and queue time. The Oracle community generally looks at service time as CPU time and queue time as everything else. This is rather convenient for tracking time because CPU time can be gathered a number of different ways from Oracle [RTA, RPM] and the queue time is recorded in Oracle's wait event views [SWA]. By combining the CPU time and the wait time, we can count time from a database server perspective (*not* end-to-end user response time).

As you can see, the general Oracle latching algorithm is actually not all that complicated. And I'm hoping you can also see that just understanding the algorithm will not empower you to solve the problem. We also need to understand the underlying Oracle architecture.

How Multiple Latches Are Implemented

Have you ever wondered why Oracle would need or could use multiple latches (e.g., library cache or cache buffer chain) if the purpose was to ensure *serial* execution of the kernel code? I mean, how could Oracle ensure one and only one process can run the kernel code if there were multiple latches? One answer is Oracle could have a master latch that would decide which child latch to run. That would work, but there is overhead which could be avoided by a simpler approach.

Another creative approach would be to simply divide the memory structure into smaller pieces and have one latch protect each piece. For example, instead of one super long LRU chain, why not have five smaller LRUs each with their own latch? Or how about instead of one single cache buffer hash chain latch covering potentially hundreds of hash buckets [RPM], use ten latches so each latch covers just ten percent of the buckets? Oracle has done this in many cases. If you ruminate on this awhile, it will force some interesting architectural questions, but they can all be answered with a correct architectural understanding.

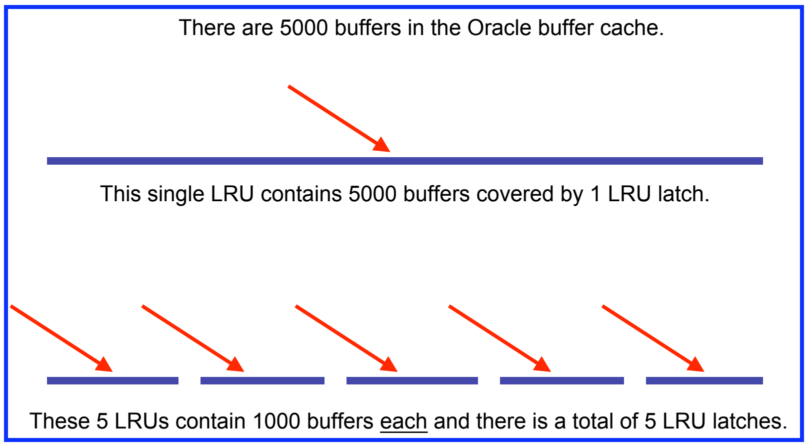

There are 5000 buffers in the Oracle buffer cache.
This single LRU contains 5000 buffers covered by 1 LRU latch.
These 5 LRUs contain 1000 buffers each and there is a total of 5 LRU latches.

Figure 2. How Oracle uses multiple latches. There are many ways Oracle could have implement multiple latches, but they chose to segment a potentially very large single LRU into potentially many small substructures, each protected by its own LRU latch.

How Time is Recorded

This is important. When a process is spinning on a latch it is consuming CPU time and when a process is sleeping it is not consuming CPU time. Even though a user experiences no difference, when a process is spinning on a latch it is recorded as CPU time and when a process is sleeping it is recorded as wait time. That is, when you see the wait event *latch free*, it is the sleep time and does not include the spinning time. So when the latch free wait event appears, you know that processes have already been trying to get a latch by spinning (which consumes CPU time) and has been forced to sleep.

With this in mind, you may be able to see why it is very common to have latch contention when there is an operating system CPU bottleneck. And also why increasing *spin_count* can make performance even worse than it already is. Why would you want to allow each process to consume even more CPU when there is not enough CPU anyway? This is why increasing *spin_count* usually makes performance worse. *Decreasing spin_count* is a valid approach to reduce latch contention.

To summarize, time can be spent in three areas:

- Getting a latch (kernel code: spin: O/S CPU time, sleep: Queue/Wait)
- Holding a latch (kernel code: CPU time)
- Releasing a latch (kernel code: CPU time)

And remember, spinning consumes CPU time, so increasing the spin will probably consume more CPU. Spinning on a latch does not post a wait event. And finally, sleeping consumes no CPU time and does post a *latch free* wait event.

How To Detect Harmful Latch Contention

Detecting Oracle latch contention is very easy. The challenge for most people is determining which latch is the real problem and if they should spend their time reducing the contention. I'll address both of these issues in this section.

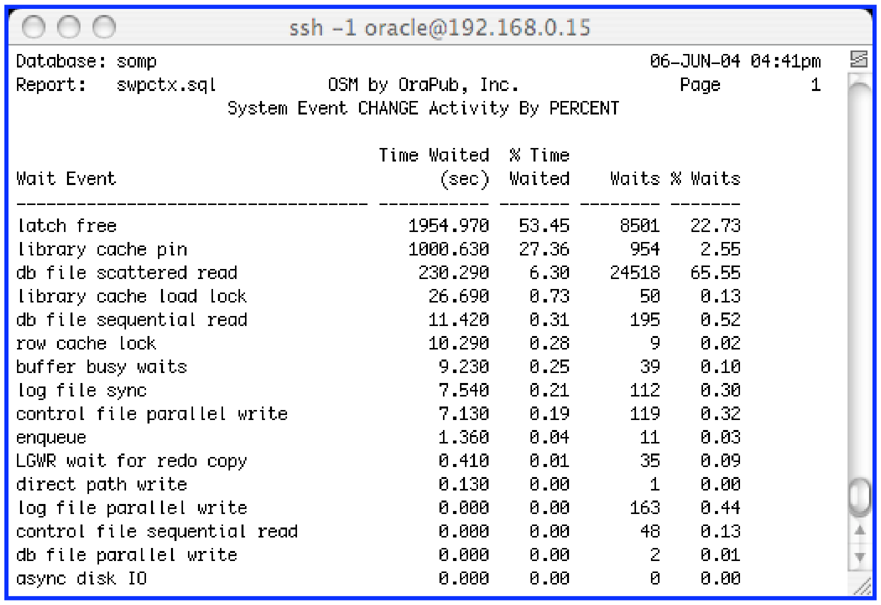
There are four possibilities. You can either be looking at contention from a system perspective or a session perspective. And you could be looking at the situation from an interactive (i.e., real time) or a historical perspective (i.e., gathered data and now reviewing). Regardless, with the right tools, you can successfully identify if there is harmful latch contention and specifically the problem latch.

For a complete explanation of how to use Oracle wait events and perform a response time analysis, please take a few minutes to review my papers *Direct Contention Identification Using Oracle's Wait Interface* [SWA] and *Oracle Response Time Analysis* [RTA]. These two are some of my most popular papers and I think you will learn quite a bit from them.

Using the session wait event views, look for the wait events that start with latch free. In Oracle 10g and higher, Oracle includes the latch name as part of the event name. If you are looking from system perspective, look at v$system_event for the event *latch free*. If the latch name is not part of the wait event name, then if you are investigating interactively, then look at v$session_wait where the event name starts with *latch free*. The latch number, which can be joined with v$latchname, is in column *P2*.

The basic SQL can be quite simple, but a more useful report is a bit more complicated. With Oracle systems not reinitializing sometimes for months at a time, a simple query from v$system_event will include perhaps thousands of hours of accumulated wait information. Important: What has been accumulated does not necessarily indicate the current contention situation. What you need is a report that captures information over a period of time. I call this a delta or a *blue line* report in my class.[RPM, TPM, OSM] For example, you could run a report now, store the start time data, wait a few minutes, and then run the report again to show the difference and re-store the start time for the next run. The output from such a report is shown in Figure 3. Identifying latch contention can be very simple. The key is to determine if the contention is significant (i.e., harmful) and the latch. This above report shows that latching is consuming 53% of Oracle process wait time. That is very significant. The next figure below will show us how to identify the specific latch. Note that in Oracle 10g and higher, the latch name is part of the event name.

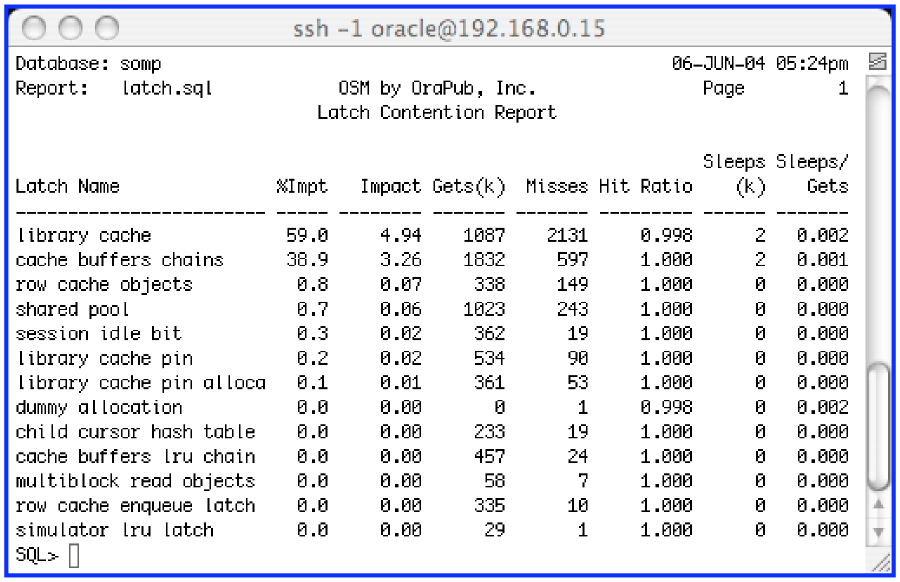
If you are pre-10g and only have access to historical latching contention data via v$system_event, you will need to use v$latch to determine which latch the processes are contending for. Figure 4 below is an example report. The *impact* columns (raw number and percentage figures) take into account the number of sleeps and is by far the most accurate statistic I have seen in showing which latch is causing the problem. This *impact* is based upon a Steve Adam's script. [http://www.ixora.com.au]



```
Database: somp                                              06-JUN-04 04:41pm
Report:   swpctx.sql          OSM by OraPub, Inc.              Page         1
                          System Event CHANGE Activity By PERCENT

                                     Time Waited  % Time
Wait Event                                 (sec)  Waited     Waits % Waits
------------------------------------ ----------- ------- --------- -------
latch free                              1954.970   53.45      8501   22.73
library cache pin                       1000.630   27.36       954    2.55
db file scattered read                   230.290    6.30     24518   65.55
library cache load lock                   26.690    0.73        50    0.13
db file sequential read                   11.420    0.31       195    0.52
row cache lock                            10.290    0.28         9    0.02
buffer busy waits                          9.230    0.25        39    0.10
log file sync                              7.540    0.21       112    0.30
control file parallel write                7.130    0.19       119    0.32
enqueue                                    1.360    0.04        11    0.03
LGWR wait for redo copy                    0.410    0.01        35    0.09
direct path write                          0.130    0.00         1    0.00
log file parallel write                    0.000    0.00       163    0.44
control file sequential read               0.000    0.00        48    0.13
db file parallel write                     0.000    0.00         2    0.01
async disk IO                              0.000    0.00         0    0.00
```

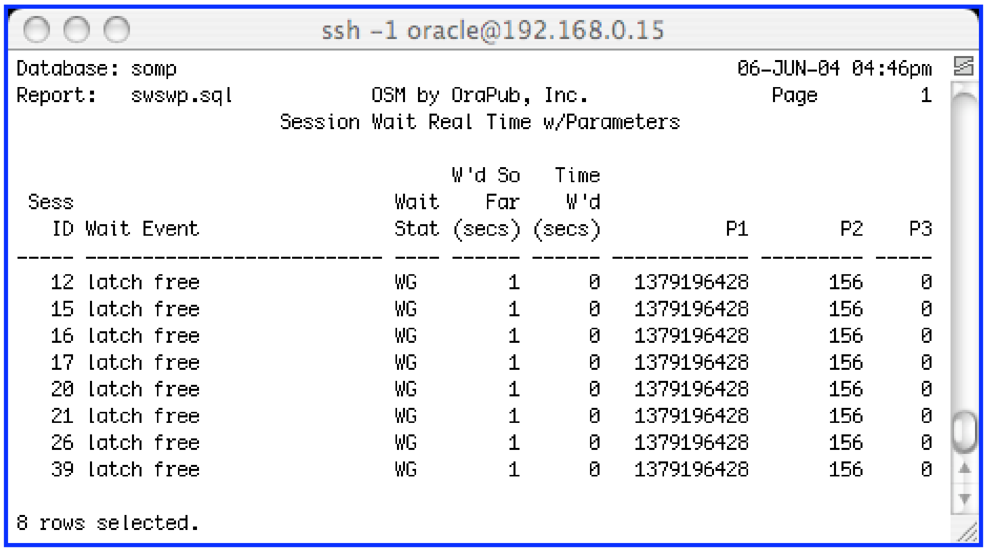
Figure 3. Identifying latch contention can be very simple. The key is to determine if the contention is significant (i.e., harmful) and the latch. This above report shows that latching is consuming 53% of Oracle process wait time. That is very significant. The next figure below will show us how to identify the specific latch. Note that in Oracle 10g and higher, the latch name is part of the event name.



```
Database: somp                                               06-JUN-04 05:24pm
Report:   latch.sql            OSM by OraPub, Inc.              Page         1
                               Latch Contention Report


                                                                    Sleeps Sleeps/
Latch Name                 %Impt    Impact Gets(k)   Misses Hit Ratio    (k)    Gets
------------------------- ----- --------- -------- -------- --------- ------ -------
library cache              59.0      4.94     1087     2131     0.998      2   0.002
cache buffers chains       38.9      3.26     1832      597     1.000      2   0.001
row cache objects           0.8      0.07      338      149     1.000      0   0.000
shared pool                 0.7      0.06     1023      243     1.000      0   0.000
session idle bit            0.3      0.02      362       19     1.000      0   0.000
library cache pin           0.2      0.02      534       90     1.000      0   0.000
library cache pin alloca    0.1      0.01      361       53     1.000      0   0.000
dummy allocation            0.0      0.00        0        1     0.998      0   0.002
child cursor hash table     0.0      0.00      233       19     1.000      0   0.000
cache buffers lru chain     0.0      0.00      457       24     1.000      0   0.000
multiblock read objects     0.0      0.00       58        7     1.000      0   0.000
row cache enqueue latch     0.0      0.00      335       10     1.000      0   0.000
simulator lru latch         0.0      0.00       29        1     1.000      0   0.000
SQL>
```

Figure 4. Identify Contending Latch using v$latch. If you do not have access to real time wait event information, such as when you are looking at historical data, you can always look at v$latch. However, identifying the problematic latch focuses on using sleeps and gets, not the classic hit ratios. Steve Adams [www.ixora.com.au] invented the *impact* calculation that has never misled me when determining the contending latch.



```
Database: somp                                                   06-JUN-04 04:46pm
Report:   swswp.sql            OSM by OraPub, Inc.                 Page         1
                          Session Wait Real Time w/Parameters

                                         W'd So   Time
 Sess                               Wait    Far    W'd
   ID Wait Event                    Stat (secs) (secs)           P1         P2    P3
----- -------------------------- ---- ------ ------ ------------ --------- -----
   12 latch free                    WG        1      0   1379196428       156     0
   15 latch free                    WG        1      0   1379196428       156     0
   16 latch free                    WG        1      0   1379196428       156     0
   17 latch free                    WG        1      0   1379196428       156     0
   20 latch free                    WG        1      0   1379196428       156     0
   21 latch free                    WG        1      0   1379196428       156     0
   26 latch free                    WG        1      0   1379196428       156     0
   39 latch free                    WG        1      0   1379196428       156     0

8 rows selected.
```

Figure 5. Identify Contending Latch using v$session_wait. If you are interactively analyzing a system, you can simply look at v$session_wait to see what sessions are currently waiting for. For the latch free wait event(s), column P2 contains the latch number, which can be joined with v$latchname. Latch number 156 is the library cache latch.

Resolving Harmful Latch Contention

Once harmful latching contention has been identified and confirmed, you obviously will want to eradicate the contention. How to do this is very specifically dependant on the contending latch. You will need to understand what the kernel code is doing that the contending latch(es) is surrounding. I have found that once I know the contending latch and I understand the underlying kernel code, the possible solutions just naturally come to mind. To help in this process, I also ask myself a few questions (see below).

In our example above, library cache latch contention is clearly the contention from an Oracle perspective. [TPM, RPM] Below are the three questions I initially ask myself.

1. *What is the library cache kernel code used for?* Very generally and incomplete, the library cache kernel code is used to check if a SQL statement is cached in the library cache.

2. *Why would a process want to run this code so often?* Because there are many, many SQL statements being run.

3. *Why would processes be holding the latch so long?* Because there are many, many unique SQL statements and therefore the hash chains (this is where/why you need to understand the underlying architecture) are relatively long. Because they are long, they take longer to sequentially scan, which means the latch must be held longer.

So the possible solutions to consider would then naturally be:

- Use bind variables.
- Enable cursor sharing
- Increase the number of library cache latches
- Increase the number of library cache buckets
- Decrease *spin_count*
- Reduce the application usage/load
- Increase the number of CPUs
- Increase the speed of CPUs

How appealing these solution possibilities are, is dependant on a number of things like ease of application modification, available CPU power, Oracle version, comfort level with changing underscore parameters, etc. But having a list of options allows communication, discussion, and *choice*…and that is what is needed in a time like this.

Conclusion

In my Oracle career, I have had the opportunity to research, write, and publish many papers. However, latching always posed a problem because the topic could be so complex and so easy to get lost in the details, I was never able to create a solid course module that I felt really good about. Through the process of repeatedly and casually teaching latching in my *Advanced Reactive Performance Management* course [RPM] combined with additional latching consulting engagements, I discovered a way to quickly teach latching to DBAs. The breakthrough came when I separated the general latching algorithm from individual latch specifics (underlying kernel code and memory structures). I could tell from my students that they immediately “got it” and were able to quickly arrive and understand many latching contention solution alternatives. As a teacher, it feels really good when that happens. How well this paper communicates that, you are the judge. But I hope you enjoyed this journey and latch contention no longer carries the stigma it may have once caused you.

About the Author

Quoted as being "An Oracle performance philosopher who has a special place in history of Oracle performance management," Mr. Shallahamer brings his unique experiences to many as a keynote speaker, a sought after teacher, a researcher and publisher for ever improving Oracle performance management, and the founder of the grid computing company, BigBlueRiver. He is a recognized authority in the Oracle server technology community and is making waves in the grid community the result of founding a company which provides "Massive grid processing power—for the rest of us."

Mr. Shallahamer spent nine years at Oracle Corporation personally impacting literally hundreds of consultants, companies, database administrators, performance specialists, and capacity planners throughout the world. He left Oracle in 1998 to start OraPub, Inc. a company focusing on "Doing and helping others Do" both reactive and proactive Oracle performance management. He continues to push performance management forward with his research, writing, consulting, highly valued teaching, and speaking engagements.

Combining his understanding of Oracle technology, the internet, and self organizing systems, Mr. Shallahamer founded BigBlueRiver in 2002 to help meet the needs of people throughout the world living in developing countries. People with limited technical and business skills can now start their own businesses which supply computing power into BigBlueRiver's computing grid. In a small way, this is making a difference in potentially thousands of people's lives.

Whether speaking at an Oracle, a grid computing, or a spiritual gathering, Mr. Shallahamer combines his experiences and his purpose toward communicating his unique insight into the technologies, the challenges, and the controversies of both Oracle and grid computing.

Reference

[RPM] "Advanced Reactive Performance Management For Oracle Based Systems" Class Notes (1998-). OraPub, Inc., http://www.orapub.com

[PPM] "Advanced Proactive Performance Management For Oracle Based Systems" Class Notes (1998-). OraPub, Inc., http://www.orapub.com

[BBR] *BigBlueRiver, Inc.*, Massive Grid Computing Power—For the Rest of Us., http://www.bigblueriver.com

[Gunther] Gunther, Neil J.; *The Practical Performance Analyst*. McGraw Hill, 2000. ISBN 0-595-12674-X

[OSM] "*OraPub System Monitor (OSM)*" tool kit (1998-)

[TC] Shallahamer, C.; *All About Oracle's Touch-Count Data Block Buffer Algorithm*. OraPub, Inc. 2001-. http://www.orapub.com

[REORG] Shallahamer, C.; *Avoiding A Database Reorganization*. OraPub, Inc. 1994. http://www.orapub.com

[SWA] Shallahamer, C.; *Direct Contention Identification Using Oracle's Session Wait Event Views*. OraPub, Inc. 1997-. http://www.orapub.com

[TRIAGE] Shallahamer, C.; *Oracle Performance Triage: Stop the Bleeding!* OraPub, Inc. 2000. http://www.orapub.com

[RTA] Shallahamer, C.; *Response Time Analysis for Oracle Based Systems.* OraPub, Inc. 2001. http://www.orapub.com

[TPM] Shallahamer, C.; *Total Performance Management.* OraPub, Inc. 1994. http://www.orapub.com