

Tuning 'log file sync' event waits

Riyaj Shamsudeen

orainternals.wordpress.com

In this blog entry, we will discuss strategies and techniques to resolve 'log file sync' waits. This entry is intended to show an approach based upon scientific principles, not necessarily a step by step guide. Let's understand how LGWR is inherent in implementing commit mechanism first.

Commit mechanism and LGWR internals

At commit time, process creates a redo record [containing commit opcodes] and copies that redo record in to log buffer. Then that process signals LGWR to write contents of log buffer. LGWR writes from log buffer to log file and signals user process back completing a commit. Commit is considered successful after LGWR write is successful.

Of course, there are minor deviation from this general concept such as latching, commits from plsql block or IMU based commit generation etc. But general philosophy still remains the same.

Signals, semaphore and LGWR

Following section introduces internal workings of commit and LGWR interaction in unix platform. There are minor implementation differences between few unix flavors or platform like NT/XP such as use of post wait drivers instead of semaphores etc. This section is to introduce internals, not necessarily dive deep in to internals. Truss is used to trace LGWR and user process to explain here.

truss command used: `truss -rall -wall -fall -vall -d -o /tmp/truss.log -p 22459`

[Word of caution, don't truss LGWR or any background process unless it is absolutely necessary. You can accidentally cause performance issues, worse yet, shutdown database.]

1. Initially, LGWR is sleeping on semaphore using `semtimedop` or `semop` call.

```
22459/1: 0.0580 0.0683 0.0000 semtimedop(9, 0xFFFFFD7FFFD7FE648, 1, 0xFFFFFD7FFFD7FE488) Err#11 EAGAIN
22459/1:          semnum=15  semop=-1  semflg=0
22459/1:          timeout: 2.060000000 sec
  In the above call,
    9 is semaphore set id visible through ipcs command and semnum=15 is the semaphore for LGWR
process in that set.
  next argument is a structure sembuf
    { unsigned short sem_num; /* semaphore number */
      short          sem_op; /* semaphore operation */
      short          sem_flg; /* operation flags */
    }
  third argument is # of semaphores
```

2. When a session commits, a redo record created and copied in to log buffer. Then that process posts LGWR semaphore using a `semctl` call, if LGWR is not active already. Then, process goes to sleep with `semtimedop` call, in its own semaphore.

Semaphore set id is 9, but `semnum` is 118 which is for the user process I was tracing.

First `semctl` calls is posting LGWR. Then process is sleeping on `semtimedop` call.

```

27396/1:      17.0781 semctl(9, 15, SETVAL, 1)                = 0
27396/1:      17.0789 semtimedop(9, 0xFFFFFD7FFDFC128, 1, 0xFFFFFD7FFDFBFB68) = 0
27396/1:      semnum=118  semop=-1  semflg=0
27396/1:      timeout: 1.000000000 sec

```

3. Waiting log writer gets a 0 return code from semtimedop and writes redo records to current redo log file. kaio calls are kernelized asynchronous I/O calls in Solaris platform.

```

22459/7:      0.2894 0.0007 0.0001 pwrite(262, "01 "\0\09E0E\0\0 i ?\0\0".., 1024, 1915904) = 1024
22459/9:      0.2894 0.0006 0.0001 pwrite(263, "01 "\0\09E0E\0\0 i ?\0\0".., 1024, 1915904) = 1024
22459/1:      0.2895 0.0007 0.0000 kaio(AIOWAIT, 0xFFFFFD7FFDFE310) = 1
22459/1:      timeout: 600.000000 sec
22459/9:      0.2895 0.0001 0.0000 kaio(AIONOTIFY, 0) = 0
22459/7:      0.2895 0.0001 0.0000 kaio(AIONOTIFY, 0) = 0

```

4. After successful completion of write(s), LGWR Posts semaphore of waiting process using semctl command.

```

22459/1:      0.2897 0.0002 0.0000 semctl(9, 118, SETVAL, 1)                = 0

```

5. User process/Session continues after receiving a return code from semtimedop call, reprinted below.

```

27396/1:      17.0789 semtimedop(9, 0xFFFFFD7FFDFC128, 1, 0xFFFFFD7FFDFBFB68) = 0

```

So, what exactly is 'log file sync' wait ?

Commit is not complete until LGWR writes log buffers including commit redo records to log files. In a nutshell, after posting LGWR to write, user or background processes waits for LGWR to signal back with 1 sec timeout. User process charges this wait time as 'log file sync' event.

In the prior section, 'log file sync' waits starts at step 2 after semctl call and completes after step 5 above.

Root causes of 'log file sync' waits

Root causes of 'log file sync', essentially boils down to few scenarios and following is not an exhaustive list, by any means!

1. LGWR is unable to complete writes fast enough for one of the following reasons:
 - a. Disk I/O performance to log files is not good enough. Even though LGWR can use asynchronous I/O, redo log files are opened with DSYNC flag and buffers must be flushed to the disk (or at least, written to disk array cache in the case of SAN) before LGWR can mark commit as complete.
 - b. LGWR is starving for CPU resource. If the server is very busy, then LGWR can starve for CPU too. This will lead to slower response from LGWR, increasing 'log file sync' waits. After all, these system calls and I/O calls must use CPU. In this case, 'log file sync' is a secondary symptom and resolving root cause for high CPU usage will reduce 'log file sync' waits.
 - c. Due to memory starvation issues, LGWR can be paged out. This can lead to slower

response from LGWR too.

d. LGWR is unable to complete writes fast enough due to file system or unix buffer cache limitations.

2. LGWR is unable to post the processes fast enough, due to excessive commits. It is quite possible that there is no starvation for cpu or memory and I/O performance is decent enough. Still, if there are excessive commits, then LGWR has to perform many writes/semctl calls and this can increase 'log file sync' waits. This can also result in sharp increase in 'redo wastage' statistics'.

3. IMU undo/redo threads. With Private strands, a process can generate few Megabytes of redo before committing. LGWR must write generated redo so far and processes must wait for 'log file sync' waits, even if redo generated from other processes is small enough.

4. LGWR is suffering from other database contention such as enqueue waits or latch contention. For example, we have seen LGWR freeze due to CF enqueue contention. This is a possible scenario however unlikely.

5. Various bugs. Oh, yes, there are bugs introducing unnecessary 'log file sync' waits.

Root cause analysis

It is worthwhile to understand and identify root cause and resolve it.

1. First make sure, 'log file sync' event is indeed a major wait events. For example in the statspack report for 60 minutes below, 'log file sync' is indeed an issue.

Why? Statspack is for 1800 seconds and there are 8 CPUs in the server. Approximately, available CPU seconds are 14,400 CPU seconds. There is just one database alone in this server and so, approximate CPU usage is 7034/14,400 : 50%. But, 27021 seconds were spent waiting. In average, $27021/3600=7.5$ processes were waiting for 'log file sync' event. So, this is a major bottleneck for application scalability.

Top 5 Timed Events

Event	waits	Time (s)	% Total Ela Time
log file sync	1,350,499	27,021	50.04
db file sequential read	1,299,154	13,633	25.25
CPU time		7,034	13.03
io done	3,487,217	3,225	5.97
latch free	115,471	1,325	2.45

2. Identify and break down LGWR wait events

a) Query wait events for LGWR. In this instance LGWR sid is 3 (and usually it is).

```
select sid, event, time_waited, time_waited_micro from v$session_event where sid=3
order by 3
SQL> /
```

SID	EVENT	TIME_WAITED	TIME_WAITED_MICRO
...			
3	control file sequential read	237848	2378480750
3	enqueue	417032	4170323279
3	control file parallel write	706539	7065393146
3	log file parallel write	768628	7686282956
3	io done	40822748	4.0823E+11€

When LGWR is waiting (using semtimedop call) for posts from the user sessions, that wait time is accounted as 'rdbms ipc message' event. This event, normally, can be ignored. Next highest waited event is 'io done' event. After submitting async I/O requests, LGWR waits until the I/O calls complete, since LGWR writes are done synchronous writes. [asynchronous and synchronous are not contradictory terms when comes to I/O! Google it and there is enormous information about this already]

It is worth to note that v\$session_event is a cumulative counter from instance startup and hence, this can be misleading. Difference between two snapshots from this view, for the same session, can be quite useful.

Tanel Poder has written an excellent <http://www.tanelpoder.com/files/scripts/snapper.sql>

Using that tool, we can print out 1 second snapshot for LGWR session 3. Some part of the output removed to improve clarity. From the table below, 813 milli seconds were spent waiting for 'io done' event in an 1 second interval. That's 81%.

SID, SNAPSHOT START	SECONDS	TYPE	STATISTIC	DELTA	D/SEC	HDELTA	HD/SEC
3, 20080513 11:44:32,	1,	STAT,	messages sent	9,	9,	9,	9
3, 20080513 11:44:32,	1,	STAT,	messages received	153,	153,	153,	153
3, 20080513 11:44:32,	1,	STAT,	redo wastage	39648,	39648,	39.65k,	39.65k
3, 20080513 11:44:32,	1,	STAT,	redo writes	152,	152,	152,	152
3, 20080513 11:44:32,	1,	STAT,	redo blocks written	1892,	1892,	1.89k,	1.89k
3, 20080513 11:44:32,	1,	STAT,	redo write time	82,	82,	82,	82
3, 20080513 11:44:32,	1,	WAIT,	rdbms ipc message	169504,	169504,	169.5ms,	169.5ms
3, 20080513 11:44:32,	1,	WAIT,	io done	813238,	813238,	813.24ms,	813.24ms
3, 20080513 11:44:32,	1,	WAIT,	log file parallel write	5421,	5421,	5.42ms,	5.42ms
3, 20080513 11:44:32,	1,	WAIT,	LGWR wait for redo copy	1,	1,	1us,	1us

3. Confirm LGWR is waiting for that event by SQL*Trace also. Tracing LGWR can deteriorate performance further. So, careful consideration must be given before turning sqltrace on LGWR. Packages such as dbms_system / dbms_support or oradebug can be used to turn on 10046 event at level 12.

Few trace lines shown below. In this specific case, LGWR is waiting for I/O events. From the output below shows that LGWR submitted 2 write calls with 16 redo blocks. Waits for I/O completion events are counted as 'io done' event and shows that between two calls, LGWR waited for 1600 micro seconds or 1.6ms. Performance of write itself is not entirely bad.

```

WAIT #0: nam='rdbms ipc message' ela= 7604 p1=223 p2=0 p3=0
WAIT #0: nam='log file parallel write' ela= 35 p1=2 p2=16 p3=2
WAIT #0: nam='io done' ela= 0 p1=0 p2=0 p3=0
WAIT #0: nam='io done' ela= 639 p1=0 p2=0 p3=0
WAIT #0: nam='io done' ela= 0 p1=0 p2=0 p3=0
WAIT #0: nam='io done' ela= 605 p1=0 p2=0 p3=0
WAIT #0: nam='io done' ela= 1 p1=0 p2=0 p3=0
WAIT #0: nam='io done' ela= 366 p1=0 p2=0 p3=0

```

4. Let's look at few other statistics also.

Statistic	Total	per Second	per Trans
redo blocks written	230,881	2,998.5	6.1
redo buffer allocation retries	0	0.0	0.0
redo entries	285,803	3,711.7	7.5
redo log space requests	0	0.0	0.0
redo log space wait time	0	0.0	0.0

redo ordering marks	0	0.0	0.0
redo size	109,737,304	1,425,159.8	2,877.5
redo synch time	40,744	529.1	1.1
redo synch writes	38,141	495.3	1.0
redo wastage	5,159,124	67,001.6	135.3
redo write time	6,226	80.9	0.2
redo writer latching time	4	0.1	0.0
user calls	433,717	5,632.7	11.4
user commits	38,135	495.3	1.0
user rollbacks	1	0.0	0.0
workarea executions - multipass	0	0.0	0.0

a. From the above statistics, we can see that 5632 user calls / second were made and 495 commits were executed per second, in average.

b. 3000 redo blocks (of size 512 bytes in solaris) is written by LGWR, approximately 1.5MB/sec.

c. 1.4MB/sec redo was generated, which is approximately 16Mbps.

d. For 3000 redo blocks, 38,155 commits were created.

Essentially, redo size is high, but not abnormal. But, 500 commits per second is on the higher side.

5. Knowledge about application will be useful here. Trace few sessions and understand why there are so many commits, if commit frequency is higher. For example, in the following trace file, there is a commit after *every* SQL statement and that can be a root cause for these waits.

XCTEND call below with rlbk=0 and rd_only=0 is a commit.

```
WAIT #120: nam='SQL*Net message from client' ela= 291 p1=1952673792 p2=1 p3=0
XCTEND rlbk=0, rd_only=0
WAIT #0: nam='log file sync' ela= 1331 p1=31455 p2=0 p3=0
WAIT #0: nam='SQL*Net message to client' ela= 1 p1=1952673792 p2=1 p3=0
```

6. If the commit frequency is higher, then LGWR could be essentially spending time signalling user process. Just like any other process, unix scheduler can kick out LGWR off the CPU and this can cause foreground process to wait for 'log file sync' event. In Solaris, prstat microstat accounting (-mL) is quite useful in breaking down how much time is spent by LGWR waiting for CPU.

7. Although uncommon, there are few bugs causing unnecessary 'log file sync' waits, signalling incorrect processes etc.

8. In few scenarios, 'log file sync' waits are intermittent and hard to catch. Statistics might need to be captured with more granularity to understand the issue. For e.g., in the example below, we had instance freeze intermittently and we wrote a small script to gather statistics from v\$sysstat every 20 seconds and spool to a file. Also, we were collecting iostat using Oracle supplied tool LTOM.

```
date : 23-MAR-2007-04:29:43 Stat: Redo blocks written : 1564176614 : Delta : 8253
date : 23-MAR-2007-04:30:14 Stat: Redo blocks written : 1564176614 : Delta : 0
date : 23-MAR-2007-04:30:44 Stat: Redo blocks written : 1564205771 : Delta : 29157
```

Between 4:29:43 and 4:30:14 redo blocks written statistics did not change since delta is 0. Meaning, LGWR did not write any redo blocks. But, 'redo size' statistics during the same duration was at ~7MB in that instance. Number of processes waiting for 'log file sync' event increased to 100. So, we know that freeze is because LGWR wasn't able to write any blocks during this time frame.

In a RAC cluster, this problem is magnified, since both CR and CUR mode buffers needs log

flush to complete before these blocks can be transferred to other instance's cache.

Looking closely at I/O statistics, we could see that average service time was quite high for few devices during that time period (we mapped these devices back to log file systems later). This lead to more granular time frame and at last, it was an hardware switch issue intermittently freezing.

extended device statistics

r/s	w/s	kr/s	kw/s	wait	actv	wsvc_t	asvc_t	%w	%b	device
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0	100	c4t50060E8003EB8200d6
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0	100	c4t50060E8003EB8200d2
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0	100	c4t50060E8003EB8200d1
0.0	0.0	0.0	0.0	0.0	9.0	0.0	0.0	0	100	c4t50060E8003EB8200d0
0.0	0.0	0.0	0.0	0.0	3.0	0.0	0.0	0	100	c4t50060E8003F9D500d13
0.0	2.0	0.0	24.0	0.0	2.0	0.0	1000.7	0	100	c4t50060E8003F9D500d12
0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0	100	c4t50060E8003F9D500d11

9. Unix tools such as truss,tusc and strace can be used to debug those scenarios if above techniques are not sufficient. But, tools such as truss and strace should be used as a last resort. In Solaris 10, dtrace can be effectively used to debug I/O or processor specific issues. Dtrace is safer by design. Brendangregg has dtrace tool kit and wealth of information here:

<http://www.brendangregg.com/dtrace.htm>

Guide lines for resolving few root causes

Finding and understanding root cause is essential to resolve a performance issue.

1. If I/O bandwith is an issue, then doing anything other than improving I/O bandwidth is not useful. Switching to file systems providing better write throughput is one option. RAW devices are another option. Reducing # of log file members in a group is another option as it reduces # of write calls. But, this option comes with a cost.

2. If CPU starvation is an issue, then reducing CPU starvation is the correct step to resolve it. Increasing priority of LGWR is a work around.

3. If commit rate is higher, then decreasing commmits is correct step but, in few case, if that is not possible, increasing priority of LGWR (using nice) or increasing priority class of LGWR to RT might provide some relief.

4. Solid State Disk devices also can be used if redo size is extreme. In that case, it is also preferable to decrease redo size. Some information can be found here.

5. If excessive redo size is root cause, redo size can be reduced using various techniques. More information can be found

http://orainternals.files.wordpress.com/2008/06/riyaj_redo_internals_tuning_by_redo_reduce_doc.pdf.

In summary, finding and resolving root cause for log file sync event, would improve application response time.