PROFILING THE LOGWRITER AND DATABASEWRITER

Frits Hoogland, Enkitec

ABSTRACT

To understand the composition of the response time of the Oracle database, most of the time sql_trace is used. This is a very valid and sane way of getting an insight into how time is spend. However, when doing DML, both the log writer and database writer(s) are key processes for the overall response time.

To gain deeper understanding of what both the log writer and database writer(s) are doing, this paper focuses on the typical wait events of these processes, and a detailed insight into what these events actually mean.

The material in the presentation and this whitepaper has been researched on Linux x86_64, Oracle Linux 6.4, and the Oracle database version 11.2.0.4. The descriptions in this paper may vary with other operating system (versions) and other Oracle database versions.

Also, at any time the COMMIT_WRITE parameter has been left default, which means the commit behavior is “immediate, wait”.

TARGET AUDIENCE

The contents of this paper are targeted at performance analysts and people who specialize in database internals. This means a general understanding of Oracle wait events, Linux, well known system calls and C program behavior are expected to be known. This paper is divided into two parts: the log writer part and the database writer part.

EXECUTIVE SUMMARY

The researcher will be able to:

- Know the common log writer and database writer wait events.
- Learn about the descriptions of the processes and wait events from the Oracle manuals.
- Learn if the Oracle provided descriptions are accurate and/or true.
- Get to know some of the implementation details of the aforementioned processes on Linux.

BACKGROUND, PART 1: THE LOGWRITER

The Oracle concepts guide describes the log writer being the process which manages the log buffer, and that its function is to write all redo entries that have been copied into the log buffer since the last time it wrote.

In idle state, the log writer waits on a System V semaphore. The semaphore can be "posted" using the semctl() system call by other processes to activate the log writer to perform its function, writing the log buffer to the online redo log files (please mind sending the log writer contents over SQL*Net is not investigated, thus not mentioned). The wait event which indicates the log writer process sleeping on a semaphore in idle state is "rdbms ipc message". The timeout of the semaphore wait is 3 seconds (seen as 300 centiseconds when looking at the "rdbms ipc message").

When the log writer is "posted" via semctl(), the general believe is it works like can be seen in picture 1:

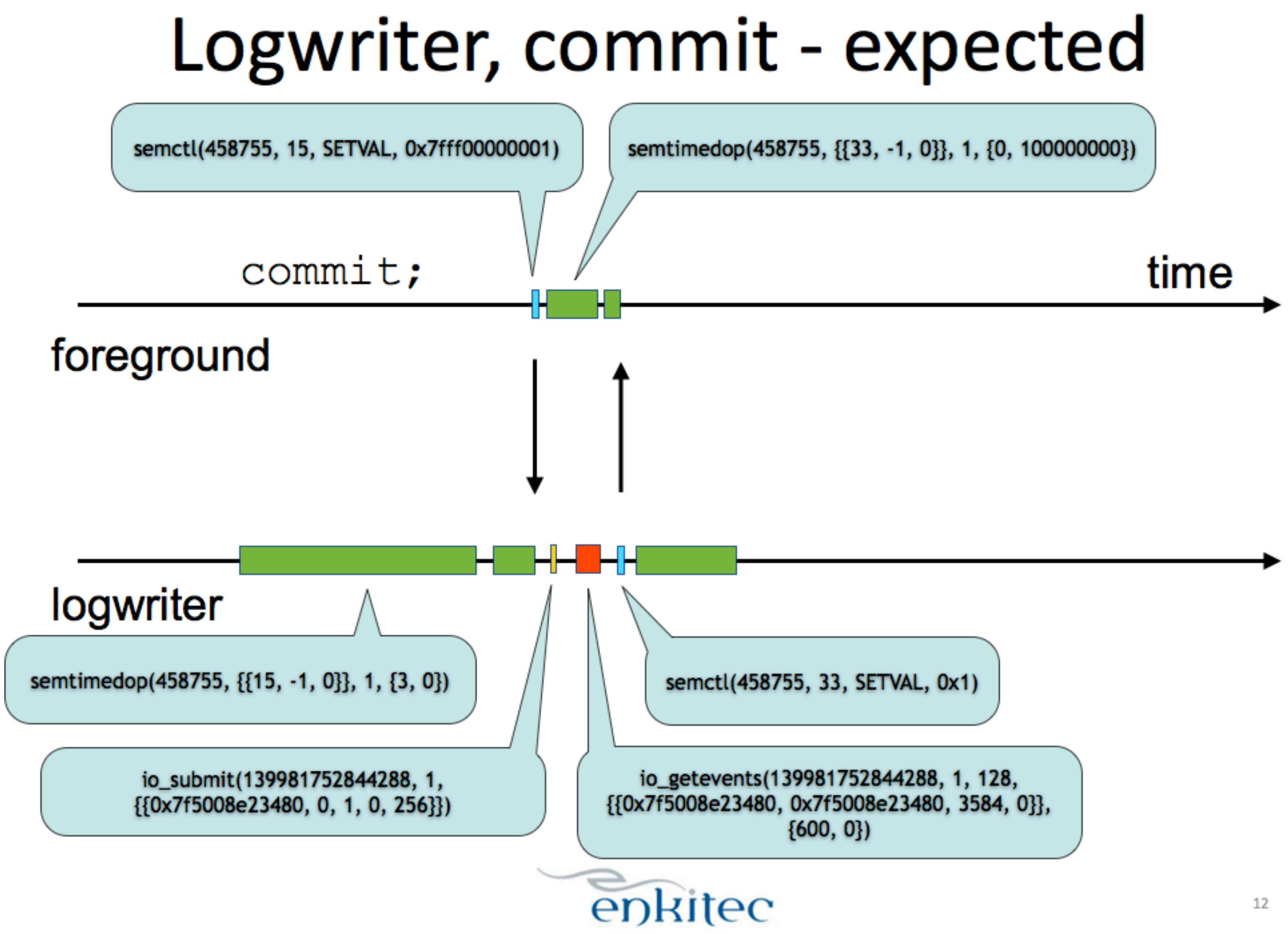

Logwriter, commit - expected
semctl(458755, 15, SETVAL, 0x7fff00000001)
semtimedop(458755, {{33, -1, 0}}, 1, {0, 100000000})
commit;
time
foreground
logwriter
semtimedop(458755, {{15, -1, 0}}, 1, {3, 0})
semctl(458755, 33, SETVAL, 0x1)
io_submit(139981752844288, 1, {{0x7f5008e23480, 0, 1, 0, 256}})
io_getevents(139981752844288, 1, 128, {{0x7f5008e23480, 0x7f5008e23480, 3584, 0}}, {600, 0})
enkitec
12

Figure 1

This picture shows a “commit”, leading to a semctl() call to post the log writer, which a-synchronously issues the IO(s; 1 in this case) and posts back the foreground process to notify the foreground process of the writing of redo information has finished.

This introduces another detail: the communication mechanism for processes in the Oracle database. The mechanism commonly used by Oracle is called “post-wait mechanism”, and uses semaphores to notify processes.

During investigation of log writer posting using the “post wait” mechanism, this is what was witnessed:

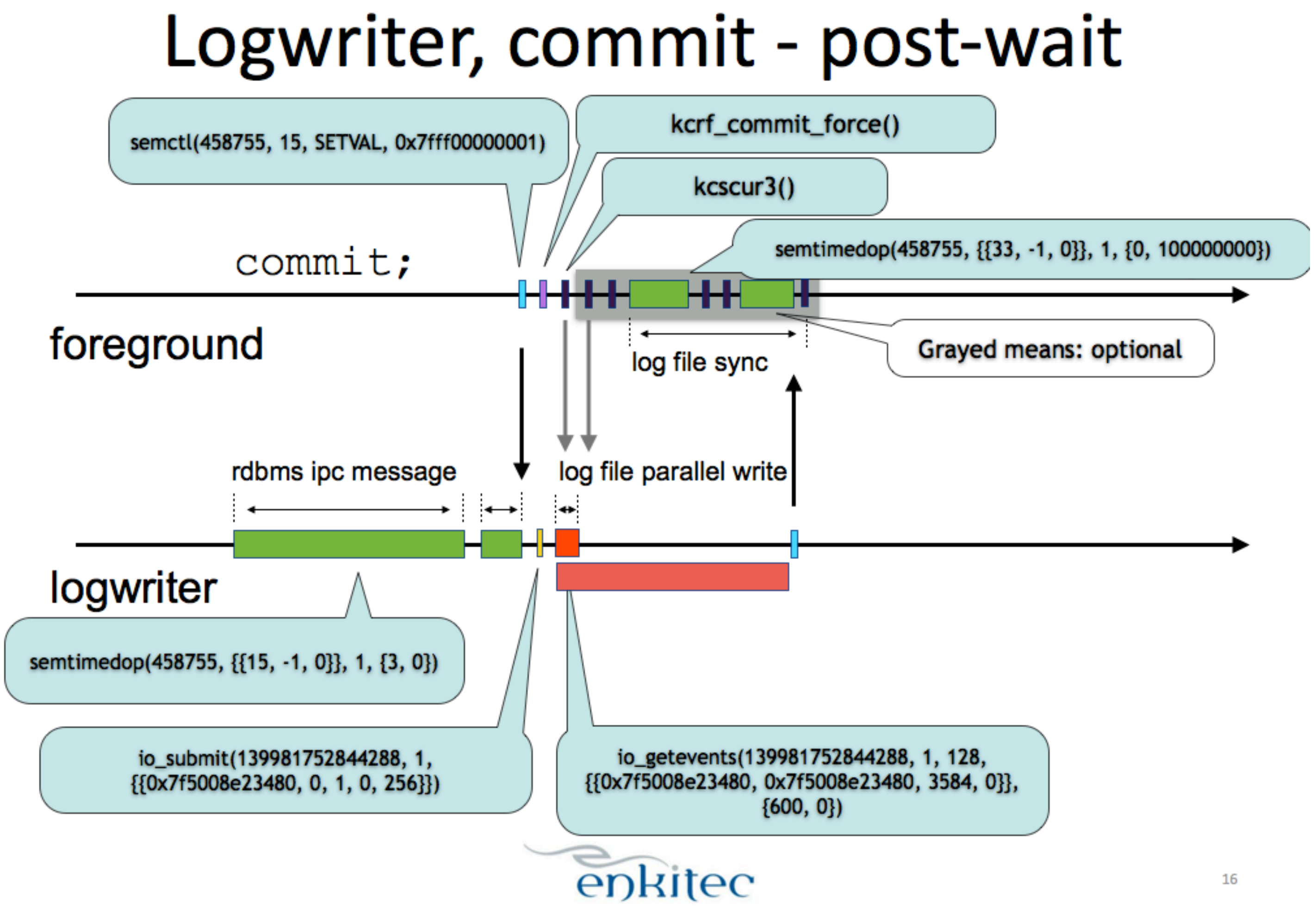

Logwriter, commit - post-wait
semctl(458755, 15, SETVAL, 0x7fff00000001)
kcrf_commit_force()
kcscur3()
semtimedop(458755, {{33, -1, 0}}, 1, {0, 100000000})
commit;
foreground
log file sync
Grayed means: optional
rdbms ipc message
log file parallel write
logwriter
semtimedop(458755, {{15, -1, 0}}, 1, {3, 0})
io_submit(139981752844288, 1, {{0x7f5008e23480, 0, 1, 0, 256}})
io_getevents(139981752844288, 1, 128, {{0x7f5008e23480, 0x7f5008e23480, 3584, 0}}, {600, 0})
enkitec
16

Figure 2

When a foreground process needs to have its generated metadata (redo) to be flushed to disk, it posts the log writer. The foreground process does not sleep on a semaphore immediately, but enters a C function called kcrf_commit_force(), and inside that function calls a function called kcscur3() up to 3 times.

It’s my strong suspicion that the kcscur3() function is used to read the log writer process progress (alias thread checkpoint SCN) to see if the log writer progressed beyond the SCN of the redo generated by the foreground. If the log writer did progress beyond the session’s needed SCN, it can skip waiting on the “log file sync” wait event altogether (which is what the greyed area indicates). If the io_getevents() call is taking a long time, which means the SCN is not updated beyond the session’s SCN after three times calling kcscur3(), the foreground session will register the “log file sync” wait, and start sleeping on a semaphore for 100ms until it gets posted by the log writer.

During this investigation I was notified that there is a second mechanism the Oracle database can use for waiting on the log writer to finish. This mechanism is called “polling”. The way foreground sessions are posted by the log writer is governed by a parameter called “_USE_ADAPTIVE_LOG_FILE_SYNC”. This parameter is introduced with Oracle version 11.2, and defaults to FALSE up to Oracle version 11.2.0.2, and starting from Oracle version 11.2.0.3, the default value of this parameter is TRUE.

With the parameter “_USE_ADAPTIVE_LOG_FILE_SYNC” set to TRUE, it means that the Oracle database chooses between “post-wait” which we just investigated, and “polling”. The database uses heuristics to make this decision. If the log file sync mode switches, the log writer writes a notification into its process trace file (thus not in any alert.log file).

Figure 3 depicts the functions found with the log file sync mode “polling”:

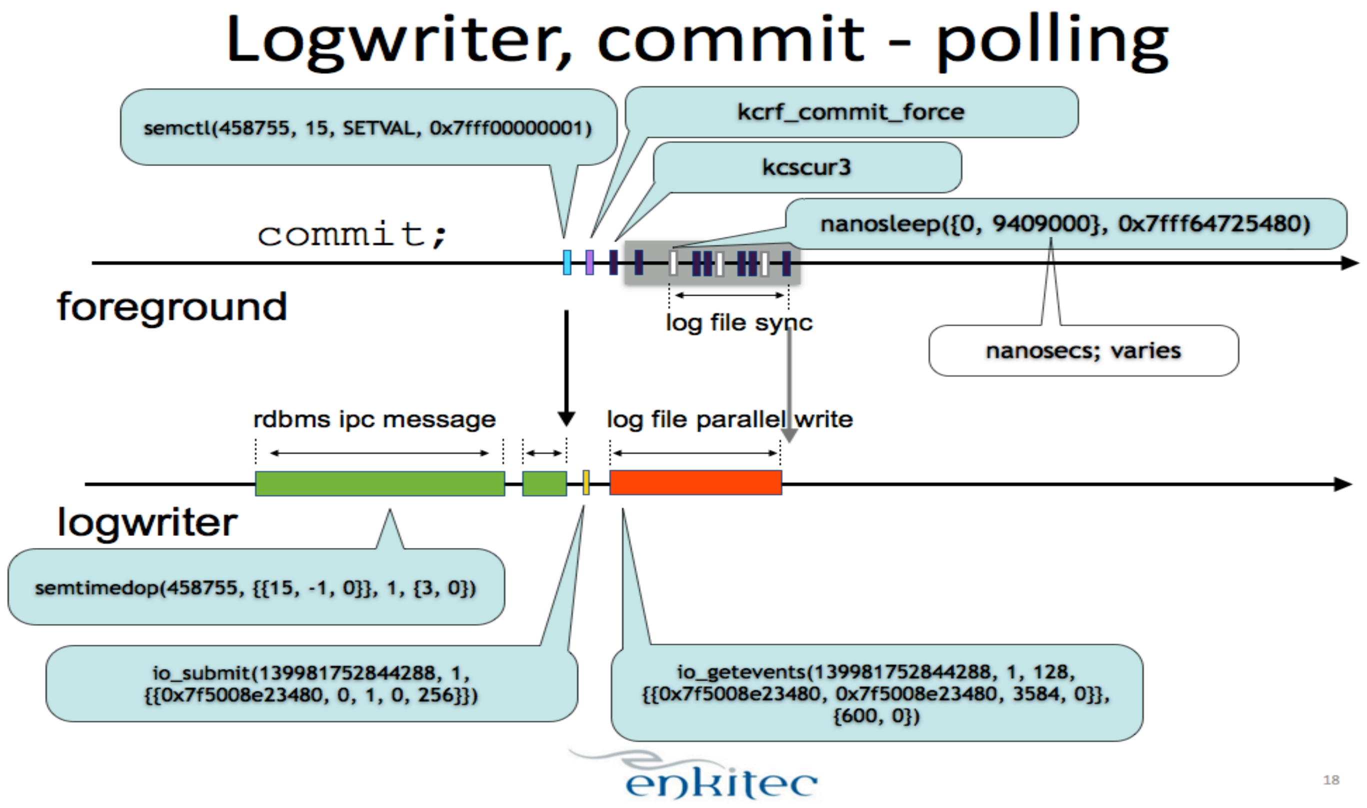
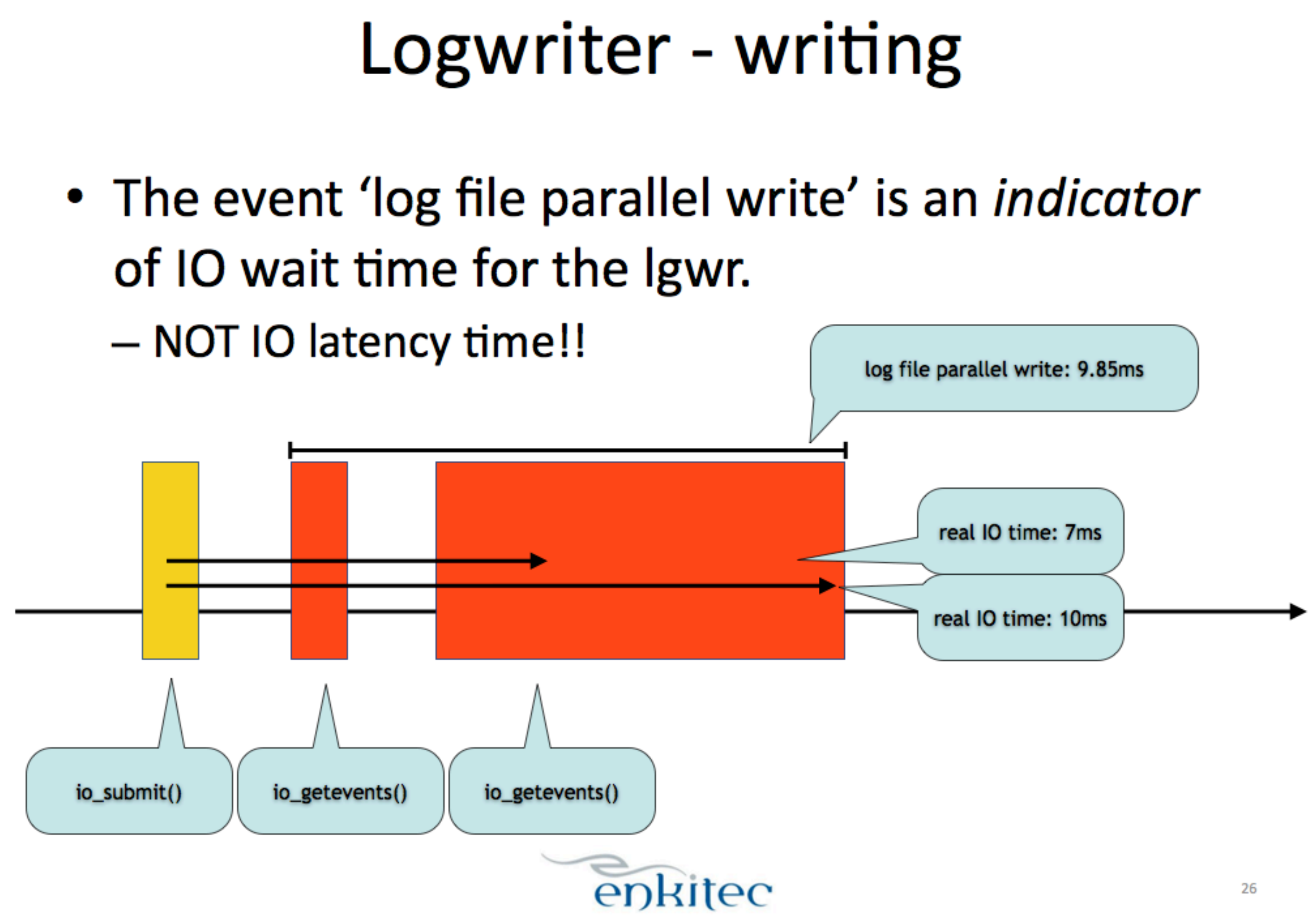

Logwriter, commit - polling
semctl(458755, 15, SETVAL, 0x7fff00000001)
kcrf_commit_force
kcscur3
nanosleep({0, 9409000}, 0x7fff64725480)
commit;
foreground
log file sync
nanosecs; varies
rdbms ipc message
log file parallel write
logwriter
semtimedop(458755, {{15, -1, 0}}, 1, {3, 0})
io_submit(139981752844288, 1, {{0x7f5008e23480, 0, 1, 0, 256}})
io_getevents(139981752844288, 1, 128, {{0x7f5008e23480, 0x7f5008e23480, 3584, 0}}, {600, 0})
enkitec
18

Figure 3

The first thing -which is important to notice- is that the notification of the log writer is still done in the original way. The “polling” log file sync mode influences how the foreground process wait for its redo to be persisted, not how it is notifying the log writer. The same functions are used as with the “post wait mechanism”, which means kcrf_commit_force() and kcscur3(), but kcscur3() is called twice instead of 3 times. If the log writer did not progress beyond the foreground’s redo SCN, it starts to sleep on the system call nanosleep(). Please mind there are no notifications by the log writer in “polling mode”, so the nanosleep() call will always be finished. It seems the time in which the process sleeps in nanosleep() is variable, and calculated from statistics, probably the same statistics as which made the database switch to polling mode in the first place.

The wait event indicating the log writer is reaping its submitted IOs is “log file parallel write”. When using asynchronous IO with the log writer, IO requests are submitted with the io_submit() system call, which is not instrumented with any wait, and the reaping of the submitted IOs is done with one or two calls. The log writer first registers the wait events “log file parallel write” (this means there is always an event “log file parallel write” when the log writer is writing to disk), then issues a non-blocking io_getevents() system call for all the submitted IOs, and if it didn’t succeed in reaping all the IOs, it issues another io_getevents() call, this time with the timeout struct set to 600 seconds, waiting for all the IOs.

Does this mean the wait event “log file parallel write” can be used to measure IO latency? No. It can be used to get an indication of how the disk subsystem of the operating system behaves, but only if the way how the wait event is timed is taken into account, and understand what can’t be seen. Figure 4 gives a simple, schematic overview:



Logwriter - writing

- The event 'log file parallel write' is an *indicator* of IO wait time for the lgwr.
  - NOT IO latency time!!

Figure 4

Figure 4 shows the simplest possible version of the log writer writing two IOs using asynchronous IO. First of all, both IOs are submitted onto the operating system IO queue. This is where the IO actually starts. Because the io_submit() system call is not timed, nor is the start of the IOs (plural) recorded in the wait interface. Immediately after the submit, the process registers the "log file parallel write" wait event, and then calls the system call io_getevents() non-blocking (which means the timeout is set to 0), asking the operating system if all IOs are finished. In this example not all the IOs are finished. If not all IOs are finished, the process issues another io_getevents() call for all IOs, but blocking, which means it just waits until the requested IOs are ready.

This means there are several things to notice here:

- There can be multiple IOs submitted at the same time.

- Because we need to wait for ALL IOs to finish, we only see the timing of the slowest IO.
- The timing of the slowest IO is actually a bit less than the actual time, because the submitting of the IO is not included in the timing.
- Because it only shows the timing of the slowest IO, any information on the timing of all the other IOs does not exist.

What happens with the wait events if we turn asynchronous IO off?

- Log file parallel write: not timing synchronous (pwrite64()) calls, which are issued serially.
- Control file sequential read: synchronous (pread64()) call is correctly timed.
- Control file parallel write: not timing synchronous (pwrite64()) calls, which are issued serially.
- Log file single write: synchronous (pwrite64()) call is correctly timed.

This means some events have serious timing issues with asynchronous IO disabled!!

If the log writer spends more than 500ms waiting on IOs ("log file parallel write"; io_getevents()) it will print a warning in the log writer process trace file (again: not in the alert logs) indicating a warning:

```
Warning: log write elapsed time 523ms, size 2760KB
```

This is governed by the hidden parameter "_SIDE_CHANNEL_TIMEOUT_MS", which defaults to 500.

Another way to influence the log writer performance is the hidden parameter "_DISABLE_LOGGING". First thing to mention is this parameter should not be used for anything, except for testing where losing the database is not a problem. Even a completely idle database generates redo, which is written to the online redo log files by the log writer and which is not written to the online redo log files when "_DISABLE_LOGGING" is set to TRUE.

Unlike popular belief, there is no magic hidden in the database when "_DISABLE_LOGGING" is set to TRUE. The foreground still posts the log writer, and the log writer still does all the same (think latching to access shared data structures like the log buffer), with the notable except of submitting (io_submit()) and reaping (io_getevents()) IOs. This means that setting "_DISABLE_LOGGING" to TRUE will only give a better performance if a significant time was spend in the system calls which it disabled (io_submit(), but more likely io_getevents()). Because io_getevents() is the most likely system call to be spending significant time in, this can actually be measured by looking at the “log file parallel write” wait event, to see what portion of the foreground “log file sync” wait event is actually backed by the log writer waiting for IO in the wait events “log file parallel write”.

This is a selective function call trace of a logwriter on Exadata:

```
kslwtbctx
semtimedop - 3309577 semid, timeout: $24 = {tv_sec = 2, tv_nsec = 970000000}
kslwtectx -- Previous wait time: 2973630: rdbms ipc message

$25 = "oss_write"
$26 = "oss_write"
$27 = "oss_write"
$28 = "oss_write"

kslwtbctx
$29 = "oss_wait"
$30 = "oss_wait"
$31 = "oss_wait"
$32 = "oss_wait"
kslwtectx -- Previous wait time: 2956: log file parallel write

kslwtbctx
semtimedop - 3309577 semid, timeout: $33 = {tv_sec = 3, tv_nsec = 0}
kslwtectx -- Previous wait time: 3004075: rdbms ipc message
```

On Exadata, Oracle uses the same wait events (“log file parallel write”,”rdbms ipc message”). The write requests on Exadata are not system calls, but normal C calls, which are “oss_write()” and “oss_wait()”. Oss_write() is the equivalent of io_submit(), and oss_wait() is the equivalent of io_getevents(). It seems that the general code path is used as much as possible, only to jump to Exadata specific code when there is a true need. This means that oss_write() is not timed, just like io_submit(), and oss_wait() is timed in the wait event “log file parallel write”.

BACKGROUND, PART 2: THE DATABASEWRITER

The Oracle concepts guide describes the database writer being a process or processes which write dirty buffers under two conditions:

- When the database writer or writers are signaled by a foreground process which cannot find a clean reusable buffer after scanning a threshold of buffers.
- The database writer or writers write buffers to advance the checkpoint, which is the position in the redo thread from which instance recovery begins.

In idle state, the database writer waits on a System V semaphore, just like the log writer. Other processes can "post" the database writer to perform writing of dirty buffers. The wait event which indicates the database writer is to be idle; means sleeping on a semaphore is "rdbms ipc message", just like the log writer.

When the database writer is traced, while a buffer or buffers are dirtied (insert a row into a table) and forced to write ("alter system checkpoint"), the sequence of wait events is:

```
WAIT #0: nam='rdbms ipc message' ela= 3000957 timeout=300 p2=0 p3=0 obj#=-1 tim=1388716667473024

*** 2014-01-03 03:37:49.735
WAIT #0: nam='rdbms ipc message' ela= 2261867 timeout=300 p2=0 p3=0 obj#=-1 tim=1388716669735046
WAIT #0: nam='db file async I/O submit' ela= 0 requests=3 interrupt=0 timeout=0 obj#=-1 tim=1388716669735493
WAIT #0: nam='db file parallel write' ela= 21 requests=1 interrupt=0 timeout=2147483647 obj#=-1
tim=1388716669735566

*** 2014-01-03 03:37:50.465
WAIT #0: nam='rdbms ipc message' ela= 729110 timeout=73 p2=0 p3=0 obj#=-1 tim=1388716670464967
```

The first and the last wait event ('rdbms ipc message') indicate that database writer is idle/sleeping. The two database writer specific wait events are 'db file async I/O submit' and 'db file parallel write'. Just by looking at the wait event names, you can derive that for the database writer, the io_submit() system call actually is instrumented as 'db file async I/O submit', and the io_getevents() system call seems to be instrumented as 'db file parallel write'.

To investigate this further, let's add the system calls to the Oracle trace information. This can be done by enabling sql_trace at level 8 for the process (which makes the process write sql_trace information and waits to its trace file), and trace the system calls using the linux 'strace' utility, and make the strace utility show the contents of the write: strace -e write=all -e all -p <PID>.

```
io_submit(140195085938688, 3, {{0x7f81b622ab10, 0, 1, 0, 256}, {0x7f81b622a8a0, 0, 1, 0, 256}, {0x7f81b622a630,
0, 1, 0, 256}}) = 3
```

```
write(13, "WAIT #0: nam='db file async I/O "..., 108) = 108
 | 00000  57 41 49 54 20 23 30 3a  20 6e 61 6d 3d 27 64 62  WAIT #0:  nam='db |
 | 00010  20 66 69 6c 65 20 61 73  79 6e 63 20 49 2f 4f 20   file as ync I/O  |
 | 00020  73 75 62 6d 69 74 27 20  65 6c 61 3d 20 31 20 72  submit'  ela= 1 r |
 | 00030  65 71 75 65 73 74 73 3d  33 20 69 6e 74 65 72 72  equests= 3 interr |
 | 00040  75 70 74 3d 30 20 74 69  6d 65 6f 75 74 3d 30 20  upt=0 ti meout=0  |
 | 00050  6f 62 6a 23 3d 2d 31 20  74 69 6d 3d 31 33 38 38  obj#=-1  tim=1388 |
 | 00060  39 37 37 36 35 31 38 30  34 32 36 31              97765180 4261     |
io_getevents(140195085938688, 1, 128, {{0x7f81b622ab10, 0x7f81b622ab10, 8192, 0}, {0x7f81b622a8a0,
0x7f81b622a8a0, 8192, 0}, {0x7f81b622a630, 0x7f81b622a630, 8192, 0}}, {600, 0}) = 3
write(13, "WAIT #0: nam='db file parallel w"..., 116) = 116
 | 00000  57 41 49 54 20 23 30 3a  20 6e 61 6d 3d 27 64 62  WAIT #0:  nam='db |
 | 00010  20 66 69 6c 65 20 70 61  72 61 6c 6c 65 6c 20 77   file pa rallel w |
 | 00020  72 69 74 65 27 20 65 6c  61 3d 20 35 38 20 72 65  rite' el a= 58 re |
 | 00030  71 75 65 73 74 73 3d 31  20 69 6e 74 65 72 72 75  quests=1  interru |
 | 00040  70 74 3d 30 20 74 69 6d  65 6f 75 74 3d 32 31 34  pt=0 tim eout=214 |
 | 00050  37 34 38 33 36 34 37 20  6f 62 6a 23 3d 2d 31 20  7483647  obj#=-1  |
 | 00060  74 69 6d 3d 31 33 38 38  39 37 37 36 35 31 38 30  tim=1388 97765180 |
 | 00070  34 35 37 39                                        4579              |
write(13, "\n", 1)                    = 1
 | 00000  0a                                                .
```

There are a lot of interesting things to see here!

- The wait event “db file async I/O submit” follows the system call io_submit(), and the wait event “db file parallel write” follows the system call io_getevents().
- The “requests” parameter with the “db file async I/O submit” wait event shows the actual number of IO requests submitted to the operating system using io_submit().
- The “requests” parameter with the “db file parallel write” wait event shows the minimal number of IO requests which are needed for the io_getevents() system call to succeed.
- The number of requests with “db file parallel write” is way lower than the actual number of IOs submitted (1 versus 3).
- The timeout value of “db file parallel write” is 2147483647. According the the Oracle documentation this is in hundredths of a second, which makes the timeout 248.55 days. That doesn’t make any sense. Especially because the io_getevents() system call has a timeout value of 600 seconds.

There is no documentation on the wait event “db file async I/O submit”, either in My Oracle Support (MOS) or in the documentation at http://docs.oracle.com.
There are a few MOS notes that mention ‘db file async I/O submit’, but these report exceptions, instead of an explanation of the wait event. An important message is in MOS note 1576956.1 (‘How to address high wait times for the direct path write temp wait event’): this event is not being tracked prior to Oracle version 11.2.0.2.

It would be logical to assume Oracle did instrument io_submit() using the "db file async I/O submit". This is NOT the case however (!!). When profiling C functions, the following pattern shows io_submit() being outside the wait event timing:

```
kslwtbctx
kslwtectx -- Previous wait time: 236317: rdbms ipc message

io_submit - 3,45e5a000 - nr,ctx
kslwtbctx
kslwtectx -- Previous wait time: 688: db file async I/O submit

kslwtbctx
io_getevents - 1,45e5a000 - minnr,ctx,timeout: $3 = {tv_sec = 600, tv_nsec = 0}
skgfr_return64 - 3 IOs returned
kslwtectx -- Previous wait time: 9604: db file parallel write
```

My guess is this is a bug in the wait event timing, because the naming of the wait event, and the values in the parameters clearly hint at it being the timing of io_submit().

Let's take a look at a description of the wait event "db file parallel write" from the Oracle documentation:

db file parallel write

This event occurs in the DBWR. It indicates that the DBWR is performing a parallel write to files and blocks. When the last I/O has gone to disk, the wait ends.

Wait Time: Wait until all of the I/Os are completed

Parameter	**Description**
requests:	This indicates the total number of I/O requests, which will be the same as blocks
interrupt:	
timeout:	This indicates the timeout value in hundredths of a second to wait for the I/O completion.

- "This indicates that the DBWR is performing a parallel write to files and blocks.": Correct, but only if asynchronous IO is enabled. With synchronous IO, a process has no option but to serially execute IO requests.
- "When the last I/O has gone to disk, the wait ends.": This is incorrect. As we saw with the wait events together with the system calls, the number of IOs it waits for is variable, and seems to be 25-33% of the total number of IOs during my testing.
- "Wait until all of the I/Os are completed": This is the same as the previous line, which is incorrect.

- "Requests: This indicates the total number of I/O requests, which will be the same as blocks": Again: incorrect, assuming that "the total number of I/O requests" means the number of requests submitted with io_submit().
- "Timeout: This indicates the timeout value is hundredths of a second to wait for the I/O completion": This is also probably not true. The O/S timeout is 600, the value 2147483647 in hundredths of a second means 248.55 days. 2147483647 is 2^32/2-1 however.

Further investigation on the system calls and the wait event "db file parallel write" show the following pattern:

- The database writer submits a number of IOs. This number is shown with the wait events "db file async I/O submit" in the "requests" parameter.
- The database writer reaps a number of IOs. The minimum number of IOs to reap is in general 25-33% of the number of submitted IOs, and is visible with the wait events "db file parallel write" in the "requests" parameter. The number can be radically vary, based on processing statistics which the database gathers during processing.
- If there are IOs left after the timed io_getevents() call, which was visible with "db file parallel write", the process issues up to 2 non-blocking io_getevents() calls for the remainder of the submitted IOs, with a maximum of 128 requests. This is not timed, nor visible at the sql_trace level.
- At this point, either new IOs are submitted, which follow the pattern starting from the first dot, or there are still non-reaped IO requests, which follows the pattern starting from the second dot ("db file parallel write"/blocking io_getevents() for a subset of the IOs), or all IOs are done, and the database writer can start sleeping on a semaphore again.

This got me thinking on what the wait event "db file parallel write" actually means. It might be clear that "db file parallel write" is not IO latency timing at all, because it only needs a subset of the total number of IOs, so it will reap the fastest ones. It can be used as an indicator of the database writer doing physical IO, but it doesn't show the actual amount of IOs it reaped, because it will reap all the IOs which it submitted which are ready with a *minimum* indicated by the "requests" parameter. This means it could have reaped anything from the minimum number in "db file parallel write" requests up to "db file async I/O submit" requests or even more if there were previously un-reaped IOs.

When looking closely at the pattern explained in the bullet-list and below, it seems to be the mechanism Oracle created for the database writer is an IO limiter, and the “db file parallel write” wait event is the timing of the IO limiting, instead of IO latency timing.

When asynchronous IO is turned off, the wait event “db file parallel write” get a little messy. With asynchronous IO turned off, the database issues pwrite64() system calls, instead of the io_submit() and io_getevents() system call pair. These calls are issued serially, because it has no other option than to submit the IOs serially. This means that wait event name “db file parallel write” is not very descriptive worded in the synchronous IO case. Of course the wait event “db file async I/O submit” does not show up anymore, because there is not submit phase anymore: pwrite64() is submit and reap in one go.

Another observation which seems to be a bug is the wait event “db file parallel write” is shown twice in the trace file for every batch of writes.

Also, when looking at the function call tracing of the waits, it becomes apparent that the synchronous IO batch which is issued is not timed:

```
kslwtbctx
semtimedop - 458755 semid, timeout: $18 = {tv_sec = 3, tv_nsec = 0}
kslwtectx -- Previous wait time: 1239214: rdbms ipc message

pwrite64 - fd, size - 256,8192
pwrite64 - fd, size - 256,8192
pwrite64 - fd, size - 256,8192

kslwtbctx
kslwtectx -- Previous wait time: 949: db file parallel write

kslwtbctx
kslwtectx -- Previous wait time: 650: db file parallel write

kslwtbctx
semtimedop - 458755 semid, timeout: $19 = {tv_sec = 1, tv_nsec = 620000000}
```

We see the “rdbms ipc message” wait event, then the write batch, in this case consisting of 3 writes, followed by the start (kslwtbctx()) and end (kslwtectx()) of two “db file parallel write” wait events, which clearly do not time the IO requests (pwrite64() lies outside of a pair of kslwtbtx() and kslwtectx()).

Given the somewhat more “troubles” we found in the database writer implementation, let’s look into how this is implemented when using Exadata:

```
kslwtbctx
semtimedop - 3309577 semid, timeout: $389 = {tv_sec = 3, tv_nsec = 0}
kslwtectx -- Previous wait time: 1266041: rdbms ipc message
```

```
$390 = "oss_write"
$391 = "oss_write"
$392 = "oss_write"
$393 = "oss_write"
$394 = "oss_write"
$395 = "oss_write"

kslwtbctx
kslwtectx -- Previous wait time: 684: db file async I/O submit

kslwtbctx
$396 = "oss_wait"
$397 = "oss_wait"
kslwtectx -- Previous wait time: 2001: db file parallel write
$398 = "oss_wait"
$399 = "oss_wait"
$400 = "oss_wait"
$401 = "oss_wait"

semctl - 3309577,23,16 - semid, semnum, cmd
kslwtbctx
semtimedop - 3309577 semid, timeout: $402 = {tv_sec = 1, tv_nsec = 630000000}
kslwtectx -- Previous wait time: 1634299: rdbms ipc message
```

Here we see (again) that the Exadata calls which replace the asynchronous IO calls io_submit() and io_getevents() are done using oss_write() to submit an IO request, and oss_wait() to reap IO requests which are ready. Again the oss_write() calls to submit IOs are not timed, then the wait event timing starts and stops for the event "db file async I/O submit", which does not seem to time anything IO related. After that two oss_wait() calls are timed (this is different from the non-Exadata code path, where there's only one io_getevents() call for a number of IOs), and the other submitted IOs are reaped outside of the timing. On Exadata, it looks like all database writer IOs are reaped after being submitted before another IO batch is submitted. Again the wait event doesn't tell anything which can help during tuning.