The Case for Sanity in Oracle's Read Event Nomenclature

CONVERGYS

Outthinking. Outdoing.

EXECUTIVE SUMMARY

Like any Oracle scientist at some point in your optimizing endeavors you have probably stared at an extended SQL trace file and pondered the reason why Oracle calls a potentially sequential disk read a *db file scattered read* and a likely random disk read a *db file sequential read*. I have read accounts for this seeming discrepancy none more interesting than Jeff Holt's assertion that these event names correspond to how blocks are stored in memory (Named Backwards,1), i.e. blocks are either stored in memory in a scattered or contiguous manner. Irrespective of the inspiration for the event nomenclature this explanation requires further lucidity; especially when confronted with a system call that reads blocks into contiguous memory but is tagged a *db file scattered read* event.

While the *db file scattered read* event name is intuitive in the context of how blocks are stored in memory, can the same be asserted for the *db file sequential read* event? If a *db file sequential read* is a single-block read isn't it inherently self-contiguous once read into virtual memory? Is it that simple? It would appear tautological to tag *db file sequential read* to a notion that an Oracle block is stored in contiguous memory. If it is that simplistic why isn't a *db file scattered read* called a *db file multi-block sequential read*? Is there another more subtle reason Oracle decided to call it a *db file scattered read*? How do contiguous memory read operations supporting *db file scattered read* events get scattered?

Other potential interpretations of this contiguous notion for *db file sequential read* events are relegated to the frameworks of successive *db file sequential read* events for a single SQL statement or multi-block *db file sequential read* events. Could most Oracle scientists conceptualize the buffer cache memory quantum as the segment block size yet struggle to reconcile this with how blocks are stored in memory, the *db file sequential read event* in particular?

This exposition into the *db file sequential read* and *db file scattered read* events is intended to elucidate the rationale for Oracle's read event nomenclature, while addressing the potential alternate interpretations of the *db file sequential read* event within the context of how blocks are stored in memory. Additionally, the mystery concerning a contiguous memory read operation that supports a *db file scattered read* event is unraveled. If you do not like to leave any stone unturned this exposition might scratch that inquisitive itch. The examples were experiments performed on a Sun Solaris 9 UNIX platform running Oracle Enterprise Edition 9.2.0.4. The UNIX system calls studied are the ubiquitous UNIX functions *pread()* and *readv()*.

AUTHOR

Eric Emrick
Senior Technical Consultant
Convergys Corporation

©2005 Convergys Corporation

THE CACHE BUFFERS LRU CHAIN AND PHYSICAL READ REQUESTS

The foundation for Oracle's physical read operations is the cache buffers LRU chain. These structures dictate the geography of the SGA (System Global Area) buffer cache and by consequence the manner in which blocks are placed into the SGA for physical read operations.

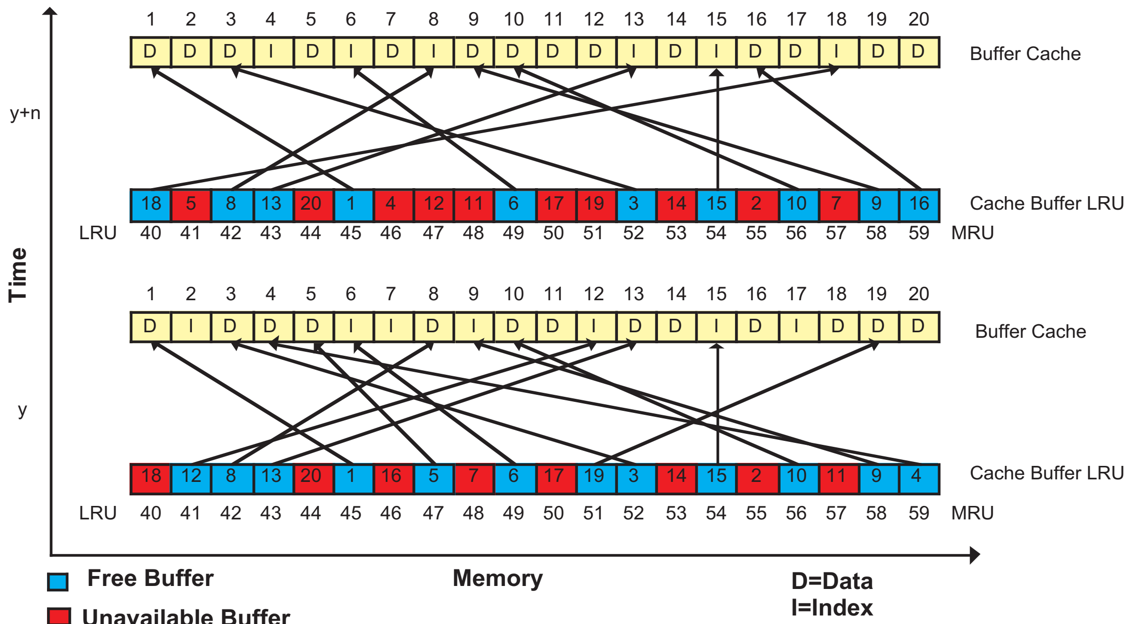

Figure 1: Buffer Cache and Cache Buffer LRU

For example, assume the Oracle kernel has a hypothetical memory allocation function called *oramalloc()*. This function traverses, from the LRU end to the MRU end, the relevant cache buffers LRU chain and returns the first found address to a free buffer in the buffer cache. For simplicity let the entire buffer cache be managed by a single cache buffers LRU chain as in Figure 1.

db file scattered read

At time t=y an interested Oracle shadow process performing a full table scan acquires the relevant cache buffers LRU chain latch. While holding the latch *oramalloc()* could be called up to eight[1] times on behalf of a *db file scattered read*. Based on the state of the SGA at time t=y, when the latch was obtained, these calls to *oramalloc()* would return the following eight free buffer addresses in sequence: 12,8,13,1,5,6,19,3. For UNIX *readv()* scatter-reads[2] these addresses are used to populate the UNIX ‹iov› address array (Stevens 404-405). This array is allocated in the calling Oracle shadow process's stack. If Oracle uses *pread()* for multi-block disk reads these same free buffer addresses are also stored in the shadow process's stack. The use of *pread()* is not synonymous with UNIX scatter-reads as a call to *pread()* only takes a single address. Nonetheless, Oracle has classified an event waiting for a multi-block read using *pread()* as a *db file scattered* read. At face value Oracle appears to have employed a sleight of hand.

The information gathered during the system call to open a file in conjunction with run time code constructs dictates if Oracle will use a call to *readv()* or *pread()* to perform the disk read that populates the buffers at the aforementioned memory addresses. The size of the *db file scattered read* (p3=?) depends on extent boundaries and blocks already in cache among other influences (Holt, Predicting,1-2).

If Oracle uses *readv()* the address array is traversed and the blocks read from disk are placed at the addresses therein. This operation is referred to as a UNIX managed scatter-read as the OS reads the blocks into the SGA. The Oracle kernel developers call this event a *db file scattered read* as it employs the read component of the longstanding UNIX scatter/gather I/O methodology.

If Oracle uses *pread()* the blocks are read from disk into the CGA (Call Global Area), a transient subheap of the PGA (Process Global Area), and then copied to the free buffer addresses stored in the shadow process's stack (Adams). Using this approach Oracle is able to simulate scatter-reads by dispatching blocks to predetermined free buffers. Oracle uses the CGA[3] as its conduit for turning a contiguous PGA memory read into a noncontiguous SGA read. This operation can be referred to as an Oracle managed scatter-read. Therefore, in spirit, the Oracle kernel developers have not violated the notion of a UNIX scatter-read and are justified in calling this multi-block *pread()* a *db file scattered read*. Notice that Oracle does not call this event scenario *a db file sequential read* with p3>1 even though the PGA has received a contiguous memory read. If the fashion in which a read system call stores data in memory has dictated the naming convention then Oracle has afforded itself latitude with its read event nomenclature.

Later at time t=y+n the same process is performing another table scan of a separate table and acquires the requisite cache buffers LRU chain latch. The next eight calls to *oramalloc()* will return the following address sequence: 18,8,13,1,6,3,15,10.

A cache buffers LRU chain manages a mutually exclusive set of memory buffers at any point in time. Each buffer's pointer position in the chain is determined by the factors that influence the heating and cooling of buffers. So it is easy to see that in a seasoned cache there is a very low probability of consecutive *oramalloc()* calls getting contiguous memory buffers for a scatter-read. The *db file scattered read* is used by Oracle to service multi-block reads that preserve physical order. The blocks for a particular segment are inspected from memory in the order they are stored on disk.

It would be terribly inefficient for *oramalloc()* to search the entire cache buffers LRU chain for contiguous memory addresses, assuming it could even find the requisite number of buffers aligned contiguously. Imagine the inefficiency of playing billiards in such a manner that your opponent is required to hit all of her shots in ball-number succession after the break has occurred, passing up perfectly accommodating and easier shots along the way.

Example 1: Several paragraphs back the use of the ephemeral CGA to accommodate *db file scattered read* events was mentioned. During my research on this topic I was intrigued by the use of *pread()* for scatter-reads given it takes a single address, thusly requiring a contiguous memory read. This notion was seemingly contrary to Oracle's buffer replacement techniques and was incongruous with the observable Oracle world. Empirically, blocks were being placed in the SGA in a scattered manner during multi-block reads but *pread()* was doing the work. What is truly fascinating is Oracle uses *pread()* to fill temporary buffers in the CGA prior to copying them to the SGA (Adams). The following experimentally demonstrates this phenomenon.

Extended SQL Trace

The FETCH and SQL*Net waits events have been excluded for brevity.

```
PARSING IN CURSOR #1   len=38  dep=0  uid=0  oct=3  lid=0  tim=889121198240  hv=3524099643
ad='19cca4e8'
select * from test_table where rownum < 5000
END OF STMT
PARSE #1: c=0,  e=215,  p=0,  cr=0,  cu=0,  mis=0,  r=0,  dep=0,  og=4,  tim=889121198233
BINDS #1:
EXEC  #1: c=0,  e=227,  p=0,  cr=0,  cu=0,  mis=0,  r=0,  dep=0,  og=4,  tim=889121198659
WAIT  #1:  nam= ' db  file  scattered  read'  ela=  18796   p1=979  p2=5    p3=8
WAIT  #1:  nam= ' db  file  scattered  read'  ela=  14318   p1=979  p2=13   p3=8
WAIT  #1:  nam= ' db  file  scattered  read'  ela=  11572   p1=979  p2=21   p3=8
WAIT  #1:  nam= ' db  file  scattered  read'  ela=  13275   p1=979  p2=29   p3=8
WAIT  #1:  nam= ' db  file  scattered  read'  ela=  11195   p1=979  p2=37   p3=8
WAIT  #1:  nam= ' db  file  scattered  read'  ela=  11036   p1=979  p2=45   p3=8
WAIT  #1:  nam= ' db  file  scattered  read'  ela=  11734   p1=979  p2=53   p3=8
WAIT  #1:  nam= ' db  file  scattered  read'  ela=  13929   p1=979  p2=61   p3=8
WAIT  #1:  nam= ' db  file  scattered  read'  ela=  12361   p1=979  p2=69   p3=8
WAIT  #1:  nam= ' db  file  scattered  read'  ela=  11247   p1=979  p2=77   p3=8
WAIT  #1:  nam= ' db  file  scattered  read'  ela=  11114   p1=979  p2=85   p3=8
WAIT  #1:  nam= ' db  file  scattered  read'  ela=  13620   p1=979  p2=93   p3=8
WAIT  #1:  nam= ' db  file  scattered  read'  ela=  11045   p1=979  p2=101  p3=8
WAIT  #1:  nam= ' db  file  scattered  read'  ela=  10976   p1=979  p2=109  p3=8
WAIT  #1:  nam= ' db  file  scattered  read'  ela=  11011   p1=979  p2=117  p3=8
STAT   #1   id=1   cnt=4999   pid=0   pos=1   obj=0   op='COUNT STOPKEY (cr=444   r=120   w=0
time=203478   us)'
STAT   #1   id=2   cnt=4999   pid=1   pos=1   obj=292723   op='TABLE ACCESS FULL TEST_TABLE
(cr=444   r=120   w=0   time=201097   us)'
```

Solaris truss

```
pread(257,  0x10339AC00,   65536,  40960)          = 65536
pread(257,  0x10339AC00,   65536,  106496)         = 65536
pread(257,  0x10339AC00,   65536,  172032)         = 65536
pread(257,  0x10339AC00,   65536,  237568)         = 65536
pread(257,  0x10339AC00,   65536,  303104)         = 65536
pread(257,  0x10339AC00,   65536,  368640)         = 65536
pread(257,  0x10339AC00,   65536,  434176)         = 65536
pread(257,  0x10339AC00,   65536,  499712)         = 65536
pread(257,  0x10339AC00,   65536,  565248)         = 65536
pread(257,  0x10339AC00,   65536,  630784)         = 65536
pread(257,  0x10339AC00,   65536,  696320)         = 65536
pread(257,  0x10339AC00,   65536,  761856)         = 65536
pread(257,  0x10339AC00,   65536,  827392)         = 65536
pread(257,  0x10339AC00,   65536,  892928)         = 65536
pread(257,  0x10339AC00,   65536,  958464)         = 65536
```

UNIX pmap

```
0000000100000000      50472K r-x--      /opt/oracle/product/9.2.0.4.0/bin/oracle
0000000103248000        712K rwx--      /opt/oracle/product/9.2.0.4.0/bin/oracle
00000001032FA000        656K rwx--           [ heap ]
0000000380000000    3530752K rwxsR      [ ism shmid=0x202 ]
FFFFFFFF7CE00000          8K rwx--           [ anon ]
FFFFFFFF7CF00000          8K r-x--      /usr/lib/sparcv9/libmd5.so.1
FFFFFFFF7D002000          8K rwx--      /usr/lib/sparcv9/libmd5.so.1
FFFFFFFF7D100000         16K r-x--      /usr/lib/sparcv9/libmp.so.2
FFFFFFFF7D204000          8K rwx--      /usr/lib/sparcv9/libmp.so.2
FFFFFFFF7D300000          8K rwx--           [ anon ]
FFFFFFFF7D400000        216K r-x--      /usr/lib/sparcv9/libm.so.1
FFFFFFFF7D534000         16K rwx--      /usr/lib/sparcv9/libm.so.1
FFFFFFFF7D600000          8K r-x--      /usr/lib/sparcv9/libkstat.so.1
FFFFFFFF7D702000          8K rwx--      /usr/lib/sparcv9/libkstat.so.1
FFFFFFFF7D800000         32K r-x--      /usr/lib/sparcv9/librt.so.1
FFFFFFFF7D908000          8K rwx--      /usr/lib/sparcv9/librt.so.1
FFFFFFFF7DA00000         32K r-x--      /usr/lib/sparcv9/libaio.so.1
FFFFFFFF7DB08000          8K rwx--      /usr/lib/sparcv9/libaio.so.1
FFFFFFFF7DB0A000          8K rwx--      /usr/lib/sparcv9/libaio.so.1
FFFFFFFF7DC00000          8K rwx--            [ anon ]
FFFFFFFF7DD00000        728K r-x--      /usr/lib/sparcv9/libc.so.1
FFFFFFFF7DEB6000         56K rwx--      /usr/lib/sparcv9/libc.so.1
FFFFFFFF7DEC4000          8K rwx--      /usr/lib/sparcv9/libc.so.1
FFFFFFFF7DF00000          8K r-x--      /usr/platform/FJSV,GPUZC-
                                        M/lib/sparcv9/libc_psr.so.1
FFFFFFFF7E000000         32K r-x--      /usr/lib/sparcv9/libgen.so.1
FFFFFFFF7E108000          8K rwx--      /usr/lib/sparcv9/libgen.so.1
FFFFFFFF7E200000        672K r-x--      /usr/lib/sparcv9/libnsl.so.1
FFFFFFFF7E3A8000         56K rwx--      /usr/lib/sparcv9/libnsl.so.1
FFFFFFFF7E3B6000         40K rwx--      /usr/lib/sparcv9/libnsl.so.1
FFFFFFFF7E400000       5328K r-x--      /opt/oracle/product/9.2.0.4.0/lib/libjox9.so
FFFFFFFF7EA32000        384K rwx--      /opt/oracle/product/9.2.0.4.0/lib/libjox9.so
FFFFFFFF7EA92000          8K rwx--      /opt/oracle/product/9.2.0.4.0/lib/libjox9.so
FFFFFFFF7EB00000         48K r-x--      /usr/lib/sparcv9/libsocket.so.1
FFFFFFFF7EC0C000         16K rwx--      /usr/lib/sparcv9/libsocket.so.1
FFFFFFFF7ED00000          8K rwx--           [ anon ]
FFFFFFFF7EE00000         32K r-x--      /opt/oracle/product/9.2.0.4.0/lib/libskgxn9.so
FFFFFFFF7EF06000          8K rwx--      /opt/oracle/product/9.2.0.4.0/lib/libskgxn9.so
FFFFFFFF7F000000          8K r-x--      /opt/oracle/product/9.2.0.4.0/lib/libskgxp9.so
FFFFFFFF7F100000          8K rwx--      /opt/oracle/product/9.2.0.4.0/lib/libskgxp9.so
FFFFFFFF7F200000          8K r-x--      /opt/oracle/product/9.2.0.4.0/lib/libodmd9.so
FFFFFFFF7F300000          8K rwx--      /opt/oracle/product/9.2.0.4.0/lib/libodmd9.so
FFFFFFFF7F400000          8K r-x--      /usr/lib/sparcv9/libdl.so.1
FFFFFFFF7F500000          8K rwx--           [ anon ]
FFFFFFFF7F600000        160K r-x--      /usr/lib/sparcv9/ld.so.1
FFFFFFFF7F726000         16K rwx--      /usr/lib/sparcv9/ld.so.1
FFFFFFFF7FFF2000         56K rw---           [ stack ]
```

Observe the virtual address *0x10339AC00* value for each *pread()* call in the *truss* output. Peruse the *pmap* sequential memory mapping for the shadow process performing the *db file scattered reads*. Notice the highlighted heap address. The repeating *pread()* address of *0x10339AC00* points into the process's heap portion of its virtual address space. The blocks read via *pread()* are headed to the PGA of the shadow process, and the CGA specifically.

db file sequential read (successive)

In the case of a single-block read the same hypothetical *oramalloc()* returns a single buffer address. The same low probability concerning contiguous memory buffers for a *db file scattered read* applies for successive *db file sequential read* events. In Figure 1 if you consider the states of the LRU chain and the buffer cache at times y and y+n, two successive *oramalloc()* calls would yield the noncontiguous free buffer addresses 12 and 18 respectively. It is often misinterpreted, within the context of how blocks are stored in memory, that successive *db file sequential reads* equate to contiguous memory storage. In Example 2 this will be demonstrated to be false in the context of the SGA using careful examination of an extended SQL trace, *truss* and buffer cache header interrogation.

Example 2: On page 8 is a snippet from an extended SQL trace of an index range scan in a database with 8K blocks. Concentration is focused on the 19 *db file sequential read* events used to service table block access as the first 3 are index blocks. While the SQL is executed a *truss* command is employed to trace just the *pread()* system calls. Finally, the buffer cache headers for the table being read are interrogated to determine the memory addresses of the blocks read from disk.

Extended SQL Trace

```
PARSING IN CURSOR   #1   len=88   dep=0   uid=0   oct=3   lid=0   tim=11170584536855
hv=753384306   ad='85193660'
select  col1,   col2  from  test_table  where  col1 = 'HGBJ'
END OF STMT
PARSE      #1: c=10000,e=6757,p=0,cr=3,cu=0,mis=1,r=0,dep=0,og=4,tim=11170584536846
BINDS      #1:
EXEC       #1: c=0, e=110, p=0, cr=0, cu=0, mis=0, r=0, dep=0, og=4, tim=11170584537201
WAIT       #1:    nam=  'SQL*Net message to client'   ela= 5       p1=1650815232   p2=1   p3=0
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 16309   p1=4    p2=3274     p3=1
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 50530   p1=23   p2=125847   p3=1
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 50813   p1=23   p2=122817   p3=1
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 43830   p1=23   p2=105746   p3=1
FETCH      #1:    c=10000, e=167997, p=4, cr=4, cu=0, mis=0, r=1, dep=0, og=4, tim=11170584705354
WAIT       #1:    nam=  'SQL*Net message from client' ela= 676     p1=1650815232   p2=1   p3=0
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 34860   p1=23   p2=105900   p3=1
WAIT       #1:    nam=  'SQL*Net message to client'   ela= 3       p1=1650815232   p2=1   p3=0
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 38847   p1=23   p2=105930   p3=1
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 31912   p1=23   p2=105592   p3=1
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 38990   p1=23   p2=105916   p3=1
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 36426   p1=23   p2=105944   p3=1
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 36074   p1=23   p2=104817   p3=1
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 33200   p1=23   p2=105561   p3=1
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 37927   p1=23   p2=110513   p3=1
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 35104   p1=23   p2=110522   p3=1
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 33214   p1=23   p2=110303   p3=1
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 31605   p1=23   p2=110454   p3=1
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 32753   p1=23   p2=110463   p3=1
FETCH      #1:    c=10000, e=424541, p=12, cr=16, cu=0, mis=0, r=15, dep=0, og=4, tim=11170585130838
WAIT       #1:    nam=  'SQL*Net message from client' ela= 1779    p1=1650815232   p2=1   p3=0
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 39668   p1=23   p2=110482   p3=1
WAIT       #1:    nam=  'SQL*Net message to client'   ela= 4       p1=1650815232   p2=1   p3=0
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 32685   p1=23   p2=110369   p3=1
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 35170   p1=23   p2=110364   p3=1
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 32338   p1=23   p2=110495   p3=1
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 36289   p1=23   p2=110429   p3=1
WAIT       #1:    nam=  'db   file   sequential   read'   ela= 29881   p1=23   p2=110458   p3=1
FETCH      #1:   c=0, e=207326, p=6, cr=11, cu=0, mis=0, r=11, dep=0,   og=4, tim=11170585340218
WAIT       #1:   nam=  'SQL*Net message from client'  ela= 2039735 p1=1650815232   p2=1   p3=0
STAT       #1    id=1    cnt=27    pid=0    pos=1    obj=5937    op='TABLE ACCESS BY INDEX ROWID
TEST_TABLE (cr=31 r=22    w=0    time=799486 us)'
STAT        #1    id=2    cnt=27    pid=1    pos=1    obj=5938    op='INDEX RANGE SCAN PK_TEST_TABLE (cr=5
r=3    w=0 time=124121    us)
```

Solaris truss

```
pread(256, "0602\0\001\0\fCA01 " u1A".., 8192, 0x01994000)      = 8192
pread(257, 0x380474000,   8192,   0x3D72E000)                   = 8192
pread(257, 0x3812AC000,   8192,   0x3BF82000)                   = 8192
pread(257, 0x381C14000,   8192,   0x33A24000)                   = 8192
pread(257, 0x38061A000,   8192,   0x33B58000)                   = 8192
pread(257, 0x38239A000,   8192,   0x33B94000)                   = 8192
pread(257, 0x3805DE000,   8192,   0x338F0000)                   = 8192
pread(257, 0x381C1C000,   8192,   0x33B78000)                   = 8192
pread(257, 0x381C9E000,   8192,   0x33BB0000)                   = 8192
pread(257, 0x3820D2000,   8192,   0x332E2000)                   = 8192
pread(257, 0x381BA0000,   8192,   0x338B2000)                   = 8192
pread(257, 0x381C84000,   8192,   0x35F62000)                   = 8192
pread(257, 0x380B4C000,   8192,   0x35F74000)                   = 8192
pread(257, 0x3805AC000,   8192,   0x35DBE000)                   = 8192
pread(257, 0x38050E000,   8192,   0x35EEC000)                   = 8192
pread(257, 0x3804E0000,   8192,   0x35EFE000)                   = 8192
pread(257, 0x381F8E000,   8192,   0x35F24000)                   = 8192
pread(257, 0x3820E2000,   8192,   0x35E42000)                   = 8192
pread(257, 0x380CDA000,   8192,   0x35E38000)                   = 8192
pread(257, 0x381C74000,   8192,   0x35F3E000)                   = 8192
pread(257, 0x380AEC000,   8192,   0x35EBA000)                   = 8192
pread(257, 0x3805BC000,   8192,   0x35EF4000)                   = 8192
```

Buffer Header Examination

SQL> select file#, dbablk, ba from x$bh where obj = 292723 order by 1,2;

```
FILE#            DBABLK          BA
----------       ----------      ----------------
23               99337           0000000382352000
23               110303          00000003805AC000
23               110364          0000000380CDA000
23               110369          00000003820E2000
23               110429          0000000380AEC000
23               110454          000000038050E000
23               110458          00000003805BC000
23               110463          00000003804E0000
23               110482          0000000381F8E000
23               110495          0000000381C74000
23               110513          0000000381C84000
23               110522          0000000380B4C000
23               105561          0000000381BA0000
23               105592          00000003805DE000
23               105746          0000000381C14000
23               104817          00000003820D2000
23               105900          000000038061A000
23               105916          0000000381C1C000
23               105930          000000038239A000
23               105944          0000000381C9E000
```

The sequence of *db file sequential read* events as they appear in the extended SQL trace and their corresponding (file#,block#) attributes are joined to the *truss* and buffer header data. The following table articulates this join.

Sequential Read Event Sequence#	Extended SQL Trace File#,Block#	X$BH.BA	truss pread() Address
1	23,105746	0000000381C14000	0x381C14000
2	23,105900	000000038061A000	0x38061A000
3	23,105930	000000038239A000	0x38239A000
4	23,105592	00000003805DE000	0x3805DE000
5	23,105916	0000000381C1C000	0x381C1C000
6	23,105944	0000000381C9E000	0x381C9E000
7	23,104817	00000003820D2000	0x3820D2000
8	23,105561	0000000381BA0000	0x381BA0000
9	23,110513	0000000381C84000	0x381C84000
10	23,110522	0000000380B4C000	0x380B4C000
11	23,110303	00000003805AC000	0x3805AC000
12	23,110454	000000038050E000	0x38050E000
13	23,110463	00000003804E0000	0x3804E0000
14	23,110482	0000000381F8E000	0x381F8E000
15	23,110369	00000003820E2000	0x3820E2000
16	23,110364	0000000380CDA000	0x380CDA000
17	23,110495	0000000381C74000	0x381C74000
18	23,110429	0000000380AEC000	0x380AEC000
19	23,110458	00000003805BC000	0x3805BC000

The *truss pread()* addresses are provided to corroborate the integrity of the x$bh.ba attribute. It is easy to see that in this case each memory address associated with each successive *db file sequential read* is not contiguous with the previous address, i.e. previous address + 8192. The idea that successive *db file sequential read* events equate to sequentially or contiguously stored memory buffers is contradicted by a single counterexample. This is very easily reproduced. More pointedly, in a mildly seasoned buffer cache of reasonable size it would be extremely improbable to find buffer addresses of successive *db file sequential read* events that are contiguous.

db file sequential read (multi-block)

In the extremely unlikely[4] event you encounter a multi-block *db file sequential read*, that reads blocks into the SGA, it would suffer the same low probability fate regarding the contiguous nature of the buffers in memory. For multi-block *db file sequential read* events the first N addresses found on the LRU chain would be used in the same manner as the Oracle or UNIX managed scatter-read, yet would not necessarily preserve physical order. Otherwise the event could not be differentiated from a *db file scattered read*.

I have encountered multi-block *db file sequential reads* that place temporary segment blocks into the PGA in Oracle7 and Oracle8. In all such cases the blocks were placed into contiguous PGA memory.

db file sequential read (contiguous memory)

The notion of contiguous memory in the context of a *db file sequential read* event can be interpreted several ways. You might be the Oracle scientist that believes the segment block size is the Oracle buffer cache memory quantum. Or, your notion might be all real memory is comprised of bits and contiguous memory is simply a stream of contiguous memory bits. Both assertions are correct and do not materially affect the notion that a *db file sequential read* stores blocks into contiguous memory. A single-block *db file sequential read* event reading into the SGA is inherently self-contiguous from either perspective. A single Oracle block, when placed in memory, is comprised of a single self-contiguous buffer or a contiguous stream of bits in memory.

CONCLUSION

The Oracle kernel developers might have labeled the *db file sequential read* and *db file scattered read* events according to how Oracle blocks are stored in memory. However, it has been exhibited that a *db file scattered read* event can perform a contiguous read into the PGA prior to copying the blocks to the SGA, which is only reconciled by Oracle managed scatter-reads. Also, it has been shown that the contiguous memory notion of a single-block *db file sequential read* requires clarification given a prevailing sense that the Oracle buffer cache quantum is the segment block size.

When the Oracle Wait Interface was introduced in Oracle7 (Shee, Deshpande and Gopalakrishnan xxv), similar to today in 10g, the overwhelming vast majority of physical read operations in Oracle databases were either single-block contiguous reads called *db file sequential read* events, or multi-block scatter-reads called *db file scattered read* events. Most Oracle scientists probably discard the nomenclature of these events for the greater purpose of understanding their implications in optimization. The newly aspiring or even seasoned Oracle scientist might stumble on the nomenclature within the context of how the blocks are stored in memory; hopefully this exposition shortens a potential sojourn in confusion on your path to Oracle optimization.

NOTES

[1] This assumes db_file_multiblock_read_count=8. A single *oramalloc()* call could have been made to return the requisite number of addresses. Since this is a hypothetical function it has been reduced to atomic operations for clarity, not performance.
[2] UNIX system programmers refer to a read from disk using *readv()* as scatter-read and is the read component of scatter/gather I/O.
[3] Refer to *Oracle8i Internal Services* by Steve Adams for an explanation of the Call Global Area.
[4] Although I have not encountered or seen documented a multi-block *db file sequential read* event placing blocks into the SGA, I will not rule it out. This case is included to exhaust the alternate perceptions of buffer continuity with *db file sequential read* events.

REFERENCES

Adams, Steve. “Re: X$BH.BA.” E-mail to Steve Adams. 12 April 2005.
Holt, Jeff. “Why are Oracle’s Read Events ‘Named Backwards’?” http://www.hotsos.com (Feb. 1, 2000).
Holt, Jeff. “Predicting Multi-Block Read Call Size.” http://www.hotsos.com (Jan. 3, 2000).
Shee, Richmond, Kirtikumar Deshpande, and K. Gopalakrishnan. Oracle Wait Interface: A Practical Guide to Performance Diagnostics and Tuning. Emeryville: McGraw-Hill/Osborne, 2004.
Stevens, Richard W. Advanced Programming in the UNIX Environment. 3rd ed. Reading: Addison-Wesley, 1992.

ACKNOWLEDGEMENTS

Special thanks to my colleagues Mike Wielonski, Darin Brown, Mike Witt and Michael Eubanks for their valuable feedback and commitment to knowledge.

CONTACT INFORMATION

PRODUCT SYSTEMS

ERIC EMRICK

eric.s.emrick@convergys.com

513 723 6944

INDUSTRY ANALYSTS

BOBBY D'ARCY

bobby.d'arcy@convergys.com

513 723 6956

TRADE MEDIA

JEFF HAZEL

jeff.hazel@convergys.com

513 723 7153

www.convergys.com

Convergys Corporation (NYSE: CVG) is a global leader in providing customer care, human resources, and billing services. Convergys combines specialized knowledge and expertise with solid execution to deliver outsourced solutions, consulting services, and software support. Clients in more than 60 countries speaking nearly 30 languages depend on Convergys to manage the increasing complexity and cost of caring for customers and employees. Convergys serves the world's leading companies in many industries including communications, financial services, technology, and consumer products.

Convergys is a member of the S&P 500 and a Fortune Most Admired Company. Headquartered in Cincinnati, Ohio, Convergys has more than 66,000 employees in 65 customer contact centers, three data centers, and other facilities in the United States, Canada, Latin America, Europe, the Middle East, and Asia. For more information visit www.convergys.com.

©2005 Convergys Corporation

For more information on our products and services, please visit www.convergys.com or call 1 800 344 3000 or 513 458 1300.

Corporate Headquarters

201 East Fourth Street
Cincinnati, Ohio 45202 USA
Tel: 513 723 7000
Fax: 513 421 8624

Regional Headquarters
Europe, Middle East & Africa

Cambourne Business Park, Cambourne
Cambridge CB3 6DN, UK
Tel: 44 1223 705000
Fax: 44 1223 705001

Latin America

CENU – Av.das Nacoes Unidas,
12.901-34 andar – Torre Norte
CEP: 04578-000 – Sao Paulo – Brasil
Tel: 55 11 5102 1800
Fax: 55 11 5102 1911

Asia Pacific

30 Cecil Street #11-08 Prudential Tower
Singapore 049712
Tel: 65 6557 2277
Fax: 65 6557 2727

© 2005 Convergys Corporation. All rights reserved. Convergys and the Convergys logo are registered trademarks of Convergys. Convergys refers to Convergys Corporation and its wholly owned subsidiaries. All other trademarks are the property of their respective owners. Every effort has been made to ensure that the information obtained in this document is true and correct at the time of going to press. However, the products and specifications and content in general described in this document are subject to continuous development and improvement and Convergys is entitled to change them at any time. Convergys cannot accept liability for any loss or damage of any nature whatsoever arising or resulting from the use of or reliance on information or particulars in this document. The information contained in this document is of a general nature. Should you require further advice for your particular business requirements, please refer to the contact details above.