Using gdb for Oracle tracing

Warning ☛ *Deep dive*

- Frits Hoogland
- Oaktable world
- Collaborate 2014, Las Vegas

```
This is the font size used for showing screen output. Be sure this is readable for you.
```

`whoami`

- Frits Hoogland
- Working with Oracle products since 1996

- Blog: http://fritshoogland.wordpress.com
- Twitter: @fritshoogland
- Email: frits.hoogland@enkitec.com
- Oracle ACE Director
- OakTable Member

enkitec

- Early bird discount expires April 15, 2014.
- Quick links:
  - Home: www.enkitec.com/e4
  - Registration: http://www.enkitec.com/e4/register
  - Location: http://www.enkitec.com/e4/location
  - Training Days Following E4: http://www.enkitec.com/e4/training-days

Goals & prerequisites

- Goal: explain attendees how I created the gdb scripts I use for profiling.
- Prerequisites:
  - Understanding of (internal) execution of C programs.
  - Understand system calls.

Using gdb for tracing

- gdb is not a one size fits all solution.
- Use the wait interface and statistics first!!

- If you need to dive further into either what Oracle is doing, or how it interacts with the operating system, gdb might be able to help you.

- gdb stops and cripples the performance of the process (“inferior”) it is attached to!

debug symbols

- gdb is made to function with an executable plus extra information to match the process execution with source-code line and understand variables.

- Oracle does NOT provide debug symbols for the Oracle database executable.

debug symbols

- gdb can use the symbol table in a non-stripped executable.
- Oracle does provide a public accessible repository of debug symbols of all it's packages: https://oss.oracle.com/ol6/debuginfo/
  - *Debuginfo packages must match executable packages 100% in version.*

About the VM

- The VM which I use for my demo's is:
  - OL6u3
- With the following debug info packages:
  - libaio-debuginfo-0.3.107-10.el6.x86_64
  - glibc-debuginfo-2.12-1.80.el6_3.6.x86_64
  - glibc-debuginfo-common-2.12-1.80.el6_3.6.x86_64

pread

- Demo: break on pread

pread

- Conclusion: we are stuck in syscall-template.S
- Perhaps there is a way to see more about the system call, I was not able to find that.
- However, there is a way to peek what is done!

pread

- In order to get more information out of the debugging of pread, I've looked into the function calling convention of Linux X86-64:
- http://en.wikipedia.org/wiki/X86_calling_conventions#x86-64_calling_conventions

pread

- The essence of the calling convention article on wikipedia:
  - For Linux X86-64, the function arguments are passed in the CPU registers:
- RDI, RSI, RDX, RCX, R8, R9, …

pread

- In order to know how pread() is called, use the manpage:

```
ssize_t pread(int fd,       void *buf, size_t count, off_t offset);

ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

RDI RSI RDX RCX

pread

- Demo: break on pread

pread

```
(gdb) break pread
Breakpoint 1 at 0x3f38a0ee20: file ../sysdeps/unix/syscall-template.S, line
82. (2 locations)
(gdb) c
Continuing.


Breakpoint 1, pread64 () at ../sysdeps/unix/syscall-template.S:82
82 T_PSEUDO (SYSCALL_SYMBOL, SYSCALL_NAME, SYSCALL_NARGS)


(gdb) print $rdi
$1 = 256
(gdb) shell ls -l /proc/10321/fd/256
lrwx------. 1 root root 64 Apr  1 10:59 /proc/10321/fd/256 -> /dev/
oracleasm/disk2
(gdb) print $rdx
$2 = 8192
```

pread

gdb macro in my toolset:

```
break pread64
  commands
    silent
    printf "pread64 - fd, size - %d,%d\n",$rdi,$rdx
    c
  end
break pwrite64
  commands
    silent
    printf "pwrite64 - fd, size - %d,%d\n",$rdi,$rdx
    c
  end
```

pread

- This means it’s possible to read the arguments in the CPU registers directly to understand the arguments of pread()/pwrite().

io_submit

- Let move to AIO!
- With AIO, the IO request and reap of the request is splitted.
- The IO request (read and write) is done with the system call io_submit().

io_submit

- Demo: break on io_submit

io_submit

```
(gdb) break io_submit
Breakpoint 2 at 0x3f38200660: file io_submit.c, line 23.
(gdb) c
Continuing.


Breakpoint 2, io_submit (ctx=0x7f583a92a000, nr=1, iocbs=0x7fffa2dd14a0) at
io_submit.c:23
23 io_syscall3(int, io_submit, io_submit, io_context_t, ctx, long, nr,
struct iocb **, iocbs)
```

io_submit

- This IO call does not get me in the syscall wrapper script syscall-template.S!
  - This code is from libaio.
  - For which I installed the debug info package!

io_submit

- This means I can see the arguments of the call:

```
Breakpoint 2, io_submit (ctx=0x7f583a92a000, nr=1, iocbs=0x7fffa2dd14a0) at
io_submit.c:23
23 io_syscall3(int, io_submit, io_submit, io_context_t, ctx, long, nr,
struct iocb **, iocbs)
```

- And investigate the arguments:

```
(gdb) print nr
$7 = 1
(gdb) print iocbs
$8 = (struct iocb **) 0x7fffa2dd14a0
```

io_submit

- It becomes interesting when we dive into the struct:

```
(gdb) print iocbs
$8 = (struct iocb **) 0x7fffa2dd14a0
```

```
(gdb) print **iocbs
$11 = {data = 0x7f5838aa31f8, key = 0, __pad2 = 0, aio_lio_opcode = 0,
aio_reqprio = 0, aio_fildes = 257, u = {c = {buf = 0x7f583889d000, nbytes =
106496, offset = 1581277184, __pad3 = 0, flags = 0, resfd = 0}, v = {vec =
0x7f583889d000, nr = 106496, offset = 1581277184}, poll = {events =
948555776, __pad1 = 32600}, saddr = {addr = 0x7f583889d000, len = 106496}}}
```

io_submit

- In this example, there is 1 IO:

```
(gdb) print nr
$13 = 1
```

- Another way to get the (to me) relevant information out of the struct:

```
(gdb) print iocbs[0].aio_fildes
$14 = 257
(gdb) print iocbs[0].u.c.nbytes
$15 = 106496
```

io_submit

- io_submit macro in my toolset:

```
break io_submit
  commands
    silent
    printf "io_submit - %d,%x - nr,ctx\n",nr,ctx
    set $c = nr-1
    while ( $c >= 0 )
     printf " fd: %d, nbytes: %d\n", iocbs[$c].aio_fildes,
iocbs[$c].u.c.nbytes
     set $c = $c - 1
    end
    c
  end
```

io_getevents

- Looking (and/or waiting) for IOs that are ready is done with the io_getevents() call.
- Here it gets a little more difficult:
  - io_getevents() *returns* the number of IOs.
  - If we break on io_getevents() we are *entering* the function.

io_getevents

- In order to break on the call io_getevents(), you need to break on io_getevents_0_4().
- There are two versions of io_getevents() in libaio:
  - compat0_1_io_getevents()
  - io_getevents_0_4()

io_getevents

- Demo: break on io_getevents

io_getevents

```
(gdb) break io_getevents_0_4
Breakpoint 3 at 0x3f38200620: file io_getevents.c, line 46.
(gdb) c
Continuing.


Breakpoint 3, io_getevents_0_4 (ctx=0x7f583a92a000, min_nr=2, nr=128,
events=0x7fffa2dd9b08, timeout=0x7fffa2ddab10) at io_getevents.c:46
46      if (ring==NULL || ring->magic != AIO_RING_MAGIC)
```

io_getevents

- The first important thing to look at is timeout:

```
(gdb) print *timeout
$16 = {tv_sec = 0, tv_nsec = 0}
```

- If timeout is set to zero, io_getevents() will just return the number of IOs.
- Regardless of min_nr!

io_getevents

- With timeout set to zero, I bumped into a little libaio optimisation (thanks Tanel):

(following slide contains the source code of io_getevents.c on https://git.fedorahosted.org/cgit/libaio.git)

io_getevents

```
int io_getevents_0_4(io_context_t ctx, long min_nr, long nr, struct io_event
* events, struct timespec * timeout)
{
   struct aio_ring *ring;
   ring = (struct aio_ring*)ctx;
   if (ring==NULL || ring->magic != AIO_RING_MAGIC)
      goto do_syscall;
   if (timeout!=NULL && timeout->tv_sec == 0 && timeout->tv_nsec == 0) {
      if (ring->head == ring->tail)
         return 0;
   }

do_syscall:
   return __io_getevents_0_4(ctx, min_nr, nr, events, timeout);
}
```

io_getevents

- This means that with:
  - the timeout struct set to zero
  - and no IOs ready

- io_getevents() will not do a system call!
- So is not visible with strace utility!

io_getevents

- With timeout set to zero, the call always returns immediately.
  - (This means) you can peek in the aio_ringbuffer yourself for ready IOs:

```
(gdb) print ring
$23 = (struct aio_ring *) 0x7f583a92a000
(gdb) print *ring
$24 = {id = 982687744, nr = 255, head = 1, tail = 3, magic = 2701791393,
compat_features = 1, incompat_features = 0, header_length = 32}

(gdb) print ring.tail-ring.head
$25 = 2
```

io_getevents

- I use breaking on io_getevents() only to indicate Oracle calling it.
  - Not to see if IOs are returned.
  - Because there is no *simple* way to know the actual number of reaped (returned) IOs.

io_getevents

- io_getevents macro in my toolset:

```
break io_getevents_0_4
  commands
    silent
    printf "io_getevents - min_nr: %d, ctx: %x, timeout { %d,%d }
\n",min_nr,ctx,timeout.tv_sec,timeout.tv_nsec
    c
end
```

io_getevents

- But is there a way to see IOs returned?

- Yes.

- If found an Oracle function that seems to be called for every IO returned by io_getevents():
  - skgfr_return64()

io_getevents

- Demo: break on skgfr_return64

io_getevents

- skgfr_return64 macro in my toolset:

```
break skgfr_return64
  commands
   silent
   printf "skgfr_return64 - %d IOs returned\n",$r10
   c
end
```

the wait interface

- The wait interface gives good insight into where an Oracle process is spending its time.
- The wait interface (its events) is not always well documented.
- Sometimes there's no documentation.
- Sometimes the documentation is wrong.
- Sometimes there are bugs.

the wait interface

- But how can you know?
- By looking what is exactly timed, and what not.

- This can be done by breaking on:
  - kslwtbctx
    - kernel service layer wait begin context *
  - kslwtectx
    - kernel service layer wait end context *

the wait interface

- Demo: profiling waits

the wait interface

```
(gdb) break kslwtbctx
Breakpoint 1 at 0x8f9a652
(gdb) commands
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
>silent
>printf "kslwtbctx\n"
>c
>end
(gdb) break kslwtectx
Breakpoint 2 at 0x8fa1334
(gdb) commands
Type commands for breakpoint(s) 2, one per line.
End with a line saying just "end".
>silent
>printf "kslwtectx\n"
```

the wait interface

```
(gdb) break kslwtectx
Breakpoint 2 at 0x8fa1334
(gdb) commands
Type commands for breakpoint(s) 2, one per line.
End with a line saying just "end".
>silent
>printf "kslwtectx\n"
>c
>end
(gdb) c
Continuing.
kslwtectx
kslwtbctx
kslwtectx
kslwtbctx
kslwtectx
```

the wait interface

- But wouldn’t it be nice if we know which waits these are?
  - Oracle does not provide debug symbols for the Oracle executable.

- Remember function call arguments are passed via CPU registers!

the wait interface

- The wait event number is passed in function:
  - kskthewt
  - In CPU register RSI

the wait interface

- This means if we add this breakpoint:

```
(gdb) break kskthewt
Breakpoint 3 at 0x8f93bf4
(gdb) commands
Type commands for breakpoint(s) 3, one per line.
End with a line saying just “end".
>silent
>printf "event#: %d\n”,$rsi
>c
>end
(gdb) c
```

the wait interface

- and remove the EOL character with kslwtectx:

```
(gdb) commands 2
Type commands for breakpoint(s) 2, one per line.
End with a line saying just "end".
>silent
>printf "kslwtectx "
>c
>end
```

- We get this output:

the wait interface

```
(gdb) c
Continuing.
kslwtectx event#: 352
kslwtbctx
kslwtectx event#: 348
kslwtbctx
kslwtectx event#: 445
kslwtbctx
kslwtectx event#: 197
kslwtbctx
kslwtectx event#: 197
kslwtbctx
kslwtectx event#: 352
kslwtbctx
kslwtectx event#: 348
kslwtbctx
```

the wait interface

- In another function gets the wait time passed:
  - Function kslwtrk_enter_wait_int
  - CPU register R13

- Add this as breakpoint:

the wait interface

```
(gdb) break kslwtrk_enter_wait_int
Breakpoint 4 at 0x8fa3c4c
(gdb) commands
Type commands for breakpoint(s) 4, one per line.
End with a line saying just "end".
>silent
>printf "time: %d ",$r13
>c
>end
(gdb)
```

- Now we get this output:

the wait interface

```
(gdb) c
Continuing.
kslwtectx time: 582529726 event#: 352
kslwtbctx
kslwtectx time: 376 event#: 348
kslwtbctx
kslwtectx time: 459 event#: 445
kslwtbctx
kslwtectx time: 904 event#: 197
kslwtbctx
kslwtectx time: 710 event#: 197
kslwtbctx
kslwtectx time: 782 event#: 352
kslwtbctx
kslwtectx time: 454 event#: 348
kslwtbctx
```

the wait interface

- In order to make this as convenient as possible for myself, I created a SQL script to create a gdb macro so the event *name* is shown too.
- I do not remember events by number.
- These numbers change (!)

the wait interface

- The result of this script is:

```
(gdb) source pw.gdb
Breakpoint 1 at 0x8f9a652
Breakpoint 2 at 0x8fa1334
Breakpoint 3 at 0x8f93bf4
Breakpoint 4 at 0x8fa3c4c
(gdb) c
Continuing.
kslwtectx -- Previous wait time: 155098556: SQL*Net message from client
kslwtbctx
kslwtectx -- Previous wait time: 401: SQL*Net message to client
kslwtbctx
kslwtectx -- Previous wait time: 604: asynch descriptor resize
kslwtbctx
kslwtectx -- Previous wait time: 1982: direct path read
```

combining

- For gdb it doesn’t matter at which points you break.
- This means the IO breakpoints (paio.gdb) can be combined with the generated waits script (pw.gdb).
- Next example with IO severely throttled.

combining

```
(gdb) c
Continuing.
kslwtectx -- Previous wait time: 65471312: SQL*Net message from client
kslwtbctx
pread64 - fd, size - 257,8192
kslwtectx -- Previous wait time: 1006627: db file sequential read
kslwtbctx
kslwtectx -- Previous wait time: 655: SQL*Net message to client
kslwtbctx
pread64 - fd, size - 257,8192
kslwtectx -- Previous wait time: 1007814: db file sequential read
```

combining

```
io_submit - 1,35230000 - nr,ctx
 fd: 257, nbytes: 106496
io_submit - 1,35230000 - nr,ctx
 fd: 257, nbytes: 122880
io_getevents - min_nr: 2, ctx: 35230000, timeout { 0,0 }
io_getevents - min_nr: 2, ctx: 35230000, timeout { 0,0 }
io_getevents - min_nr: 2, ctx: 35230000, timeout { 0,0 }
kslwtbctx
io_getevents - min_nr: 1, ctx: 35230000, timeout { 600,0 }
skgfr_return64 - 1 IOs returned
kslwtectx -- Previous wait time: 943333: direct path read
```

other macros

• create_pw.sql	Create pw.sql
• create_pl.sql	Create pl.sql
• pw.gdb	print wait events
• pl.gdb	print latches
• paio.gdb	print IO syscalls
• pstruct.gdb	PARSE/EXEC/FETCH
• prowsources.gdb	print rowsource ops.