Profiling Oracle with gdb
Hands on!

- Frits Hoogland

- 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: learn attendees about using gdb for profiling C function sequences.
- Prerequisites:
  - Understanding of (internal) execution of C programs.
  - Understanding of Oracle tracing mechanisms.

Disclaimer

The software for this presentation has been provided as part of Enkitec's Oracle Platinum Partner agreement. Access to the programs is limited to demonstration purposes only. Any use of these programs beyond this demonstration requires you to obtain the applicable Oracle license.

Copy the virtual box VM

- The virtual box importable appliance should be distributed (USB HDD/Thumbdrive)
- See handout for virtual machine/appliance instructions and requirements.

1. Tracing

- In order to troubleshoot a situation, the right tool needs to be selected.
  - Depending on that outcome, further investigation can be done, possibly at different layers with different tools.

- The most logical way to start troubleshoot Oracle database processing is using the wait interface and database statistics.

1. Tracing

- Hands on:
  - logon to the VM as oracle
  - logon to Oracle:

```
rs ts/ts@//localhost/v11204
```

– trace ‘select count(*) from t2’:

```
select count(*) from t2; --this is prewarming the caching of the VM disk
alter session set events ‘sql_trace level 8’;
select count(*) from t2;
alter session set events ‘sql_trace off’;
```

1. Tracing

- Copy the resulting trace file to ~/1.trc
  - When looking in the trace file
    - if you've got a fast disk
      - a low # of 'direct path read' waits is visible
  - Look at waits + fetch line (p+cr)

1. Tracing

- What is happening?
  - Obviously, there is PIO information missing.
- The next logical step is to include system call (regular IO is done using a system call) information with the trace information.

1. Tracing

- This is done using the linux utility ‘strace’.

- First flush the buffer cache:

```
sq
alter system flush buffer_cache;
```

- Then start a sqlplus session to trace:

```
rs ts/ts@//localhost/v11204
```

- Start a root (shell) session

1. Tracing

- Enable sql_trace at level 8 in the sqlplus session:

```
alter session set events ‘sql_trace level 8’
```

- Find the foreground in the root session:

```
ps -ef | grep [L]OCAL=NO
```

- And strace it with verbosity on the write call:

```
strace -e write=all -e all -p PID -o ~oracle/2.trc
```

- Start a root (shell) session.

1. Tracing

- Now execute the count(*) again in the sqlplus session.

```
select count(*) from t2;
```

- After the count(*) is finised, press CTRL-c in the strace session.

- And exit the sqlplus session.

```
exit
```

1. Tracing

- Now open the strace file in the oracle session:

```
less 2.trc
```

- And skip to the start of the execution of the count(*) statement:

```
/count
```

1. Tracing

```
statfs("/u01/app/oracle/oradata/V11204/o1_mf_ts_9jvyr69l_.dbf",
{f_type="EXT2_SUPER_MAGIC", f_bsize=4096, f_blocks=2162434, f_bfree=1108067,
f_bavail=992458, f_files=558624, f_ffree=504033, f_fsid={-602411700, -1491991187},
f_namelen=255, f_frsize=4096}) = 0

open("/u01/app/oracle/oradata/V11204/o1_mf_ts_9jvyr69l_.dbf", O_RDWR|O_DSYNC|
O_DIRECT) = 10

getrlimit(RLIMIT_NOFILE, {rlim_cur=64*1024, rlim_max=64*1024}) = 0

fcntl(10, F_DUPFD, 256)                 = 257

close(10)                               = 0

fcntl(257, F_SETFD, FD_CLOEXEC)         = 0

fstatfs(257, {f_type="EXT2_SUPER_MAGIC", f_bsize=4096, f_blocks=2162434,
f_bfree=1108067, f_bavail=992458, f_files=558624, f_ffree=504033,
f_fsid={-602411700, -1491991187}, f_namelen=255, f_frsize=4096}) = 0

write(8, "WAIT #140425323829824: nam='Disk"..., 130) = 130
 | 00000  57 41 49 54 20 23 31 34  30 34 32 35 33 32 33 38  WAIT #14 04253238 |
 | 00010  32 39 38 32 34 3a 20 6e  61 6d 3d 27 44 69 73 6b  29824: n am='Disk |
 | 00020  20 66 69 6c 65 20 6f 70  65 72 61 74 69 6f 6e 73   file op erations |
 | 00030  20 49 2f 4f 27 20 65 6c  61 3d 20 38 34 39 20 46   I/O' el a= 849 F |
 | 00040  69 6c 65 4f 70 65 72 61  74 69 6f 6e 3d 32 20 66  ileOpera tion=2 f |
 | 00050  69 6c 65 6e 6f 3d 35 20  66 69 6c 65 74 79 70 65  ileno=5  filetype |
```

1. Tracing

```
pread(257, "#\242\0\0\202]@\1P&\6\0\0\0\1\4\207\31\0\0\0\0\0\0\0\0\0\0\0\0"...,
8192, 196100096) = 8192
write(8, "WAIT #140425323829824: nam='db f"..., 124) = 124
 | 00000  57 41 49 54 20 23 31 34  30 34 32 35 33 32 33 38  WAIT #14 04253238 |
 | 00010  32 39 38 32 34 3a 20 6e  61 6d 3d 27 64 62 20 66  29824: n am='db f |
 | 00020  69 6c 65 20 73 65 71 75  65 6e 74 69 61 6c 20 72  ile sequ ential r |
 | 00030  65 61 64 27 20 65 6c 61  3d 20 39 36 34 30 20 66  ead' ela = 9640 f |
| 00040  69 6c 65 23 3d 35 20 62  6c 6f 63 6b 23 3d 32 33  ile#=5 b lock#=23 |
| 00050  39 33 38 20 62 6c 6f 63  6b 73 3d 31 20 6f 62 6a  938 bloc ks=1 obj |
| 00060  23 3d 32 30 30 34 35 20  74 69 6d 3d 31 33 39 32  #=20045  tim=1392 |
| 00070  38 39 37 31 39 34 37 31  33 30 33 31              89719471 3031     |
| 00080  38 36                                            86                |
```

1. Tracing

```
mmap(NULL,524288,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_NORESERVE,6,0x6d000) = 0x7fb7516c2000
mmap(0x7fb7516c2000,65536,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_FIXED,6, 0) = 0x7fb7516c2000
mmap(NULL,1114112,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_NORESERVE,6,0xed000) = 0x7fb7515b2000
mmap(0x7fb7515b2000,1114112,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_FIXED,6,0) = 0x7fb7515b2000
io_submit(140425370206208, 1, {{0x7fb7516c45e8, 0, 0, 0, 257}}) = 1
mmap(NULL,1114112,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_NORESERVE,6,0x1fd000)= 0x7fb7514a2000
mmap(0x7fb7514a2000,1114112,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_FIXED,6,0) = 0x7fb7514a2000
io_submit(140425370206208, 1, {{0x7fb7516c4bc0, 0, 0, 0, 257}}) = 1
io_getevents(140425370206208,1,128,{{0x7fb7516c45e8,0x7fb7516c45e8,106496,0}}, {600, 0}) = 1
write(8, "WAIT #140425323829824: nam='dire"..., 130) = 130
 | 00000  57 41 49 54 20 23 31 34  30 34 32 35 33 32 33 38  WAIT #14 04253238 |
 | 00010  32 39 38 32 34 3a 20 6e  61 6d 3d 27 64 69 72 65  29824: n am='dire |
 | 00020  63 74 20 70 61 74 68 20  72 65 61 64 27 20 65 6c  ct path  read' el |
 | 00030  61 3d 20 31 31 30 31 20  66 69 6c 65 20 6e 75 6d  a= 1101  file num |
 | 00040  62 65 72 3d 35 20 66 69  72 73 74 20 64 62 61 3d  ber=5 fi rst dba= |
 | 00050  32 33 39 33 39 20 62 6c  6f 63 6b 20 63 6e 74 3d  23939 bl ock cnt= |
 | 00060  31 33 20 6f 62 6a 23 3d  32 30 30 34 35 20 74 69  13 obj#= 20045 ti |
 | 00070  6d 3d 31 33 39 32 38 39  37 31 39 34 37 31 35 35  m=139289 71947155 |
 | 00080  35 33                                            53                |
```

1. Tracing

- So, it looks like when IOs are reaped*, Oracle registers a wait.
  - But that isn’t consistent with the low number of ‘direct path read’ waits…

- Let’s look further in 2.trc to the AIO calls:

1. Tracing

```
io_submit(140425370206208, 1, {{0x7fb7516c45e8, 0, 0, 0, 257}}) = 1
io_getevents(140425370206208,2,128,{{0x7fb7516c4bc0,0x7fb7516c4bc0,1032192,0}}, {0, 0}) = 1
io_submit(140425370206208, 1, {{0x7fb7516c4bc0, 0, 0, 0, 257}}) = 1
io_getevents(140425370206208,2,128,{{0x7fb7516c45e8,0x7fb7516c45e8,1032192, 0}}, {0, 0}) = 1
io_submit(140425370206208, 1, {{0x7fb7516c45e8, 0, 0, 0, 257}}) = 1
io_getevents(140425370206208,1,128,{{0x7fb7516c4bc0,0x7fb7516c4bc0,1032192,0}}, {600,0}) = 1
write(8, "WAIT #140425323829824: nam='dire"..., 130) = 130
| 00000  57 41 49 54 20 23 31 34  30 34 32 35 33 32 33 38  WAIT #14 04253238 |
| 00010  32 39 38 32 34 3a 20 6e  61 6d 3d 27 64 69 72 65  29824: n am='dire |
| 00020  63 74 20 70 61 74 68 20  72 65 61 64 27 20 65 6c  ct path  read' el |
| 00030  61 3d 20 36 30 35 20 66  69 6c 65 20 6e 75 6d 62  a= 605 f ile numb |
| 00040  65 72 3d 35 20 66 69 72  73 74 20 64 62 61 3d 32  er=5 fir st dba=2 |
| 00050  35 34 37 34 20 62 6c 6f  63 6b 20 63 6e 74 3d 31  5474 blo ck cnt=1 |
| 00060  32 36 20 6f 62 6a 23 3d  32 30 30 34 35 20 74 69  26 obj#= 20045 ti |
| 00070  6d 3d 31 33 39 32 38 39  37 31 39 34 37 34 32 36  m=139289 71947426 |
| 00080  38 31                                            81                |
```

1. Tracing

- Okay, for some reason:
  - some reaped IOs do result in a wait line,
  - and some other’s do not.

- We can safely assume it has to do with the IO speed, but how is the decision made?

1. Tracing

- In order to find out, we need to look at the sequence of function calls in(side) the database. This is done using gdb.

- Create a root (shell) session for gdb, and an oracle (shell) session for sqlplus (rs ts/ts@// localhost/v11204).

1. Tracing

- In the root session
- grep the PID of the foreground session:

```
ps -ef | grep [L]OCAL=NO
```

- Attach to the session with gdb:

```
gdb -p PID
```

- Enter the commands on the next slide in gdb:

1. Tracing

```
set pagination off
set logging file /home/oracle/3.trc
set logging on
break io_submit
break 'io_getevents@plt'
rbreak ^kslwt[be]ctx
commands 1-4
c
end
```

1. Tracing

- This has set 4 breakpoints in gdb:

```
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00007fcaa5083660 <io_submit>
        c
2       breakpoint     keep y   0x00000000009e5b58 <io_getevents@plt>
        c
3       breakpoint     keep y   0x0000000008c18680 <kslwtbctx+4>
        c
4       breakpoint     keep y   0x0000000008c1f92c <kslwtectx+4>
        c
```

1. Tracing

- In gdb, issue ‘c’ to continue the foreground session:
- In sqlplus, execute:

```
select count(*) from t2;
```

1. Tracing

- Once the select count(*) is ready:
  - Stop gdb (CTRL-c, q)
  - Exit sqlplus, and read 3.trc using less:

```
less 3.trc
```

1. Tracing

```
Breakpoint 5, 0x0000000007cfb4f2 in kslwtectx ()

Breakpoint 4, 0x0000000007cf47b6 in kslwtbctx ()

Breakpoint 5, 0x0000000007cfb4f2 in kslwtectx ()

Breakpoint 1, 0x00007fa883926660 in io_submit () from /lib64/libaio.so.1

Breakpoint 1, 0x00007fa883926660 in io_submit () from /lib64/libaio.so.1

Breakpoint 2, 0x000000000082d7d8 in io_getevents@plt ()

Breakpoint 1, 0x00007fa883926660 in io_submit () from /lib64/libaio.so.1

Breakpoint 1, 0x00007fa883926660 in io_submit () from /lib64/libaio.so.1

Breakpoint 2, 0x000000000082d7d8 in io_getevents@plt ()
```

1. Tracing

- This shows:
  - Oracle ending the timing of a wait (kslwtectx())
  - Then another wait is encountered (kslwtbctx()-kslwtectx())
  - Then IOs are submitted (io_submit()) and IOs are reaped (io_getevents())
  - In this example: without a wait event!

1. Tracing

- In my case, I didn’t get a single wait accompanying IO (io_submit()-io_getevents()) *because I turned virtual box disk caching on.*

- Let’s slow the IOs down to get the ‘direct path read’ waits back.

1. Tracing

- Startup a sqlplus session again (as oracle).

```
rs ts/ts@//localhost/v11204
```

- (as root) Now setup IO throttling using cgroups:

```
mkdir -p /cgroup/blkio
mount -t cgroup -o blkio none /cgroup/blkio
cgcreate -g blkio:/iothrottle
cgset -r blkio.throttle.read_iops_device="8:0 1" iothrottle
```

1. Tracing

- Find the PID of the server process:

```
ps -ef | grep [L]OCAL=NO
```

- And throttle the Oracle process:

```
echo PID > /cgroup/blkio/iothrottle/tasks
```

- Enter gdb again, and setup the breakpoints

```
gdb -p PID
```

1. Tracing

```
set pagination off
set logging file /home/oracle/4.trc
set logging on
break io_submit
break 'io_getevents@plt'
rbreak ^kslwt[be]ctx
commands 1-4
c
end
```

1. Tracing

- Make the server session runnable again in gdb by executing ‘c’ for continue.
- Now run the query again:

```
select count(*) from t2;
```

- Once the query is finished, quit gdb again (CTRL-c and q), quit sqlplus, and look in 4.trc with less.

1. Tracing

```
Breakpoint 5, 0x0000000007cfb4f2 in kslwtectx ()
Breakpoint 4, 0x0000000007cf47b6 in kslwtbctx ()
Breakpoint 5, 0x0000000007cfb4f2 in kslwtectx ()
Breakpoint 4, 0x0000000007cf47b6 in kslwtbctx ()
Breakpoint 5, 0x0000000007cfb4f2 in kslwtectx ()
Breakpoint 1, 0x00007f40a05c3660 in io_submit () from /lib64/libaio.so.1
Breakpoint 1, 0x00007f40a05c3660 in io_submit () from /lib64/libaio.so.1
Breakpoint 2, 0x000000000082d7d8 in io_getevents@plt ()
Breakpoint 2, 0x000000000082d7d8 in io_getevents@plt ()
Breakpoint 4, 0x0000000007cf47b6 in kslwtbctx ()
Breakpoint 2, 0x000000000082d7d8 in io_getevents@plt ()
Breakpoint 5, 0x0000000007cfb4f2 in kslwtectx ()
Breakpoint 1, 0x00007f40a05c3660 in io_submit () from /lib64/libaio.so.1
Breakpoint 2, 0x000000000082d7d8 in io_getevents@plt ()
Breakpoint 2, 0x000000000082d7d8 in io_getevents@plt ()
Breakpoint 4, 0x0000000007cf47b6 in kslwtbctx ()
Breakpoint 2, 0x000000000082d7d8 in io_getevents@plt ()
Breakpoint 5, 0x0000000007cfb4f2 in kslwtectx ()
```

1. Tracing

- This is the end of chapter 1.
  - In this chapter you have learned:
    - How to start a sql_trace
    - How to combine strace with sql_trace
    - How to use gdb to break and continue on C functions, in order to profile.

2. strace

- strace shows (only) system calls.
  - This means calls which require kernel interaction from the user land process.
  - Things like: disk I/O, network I/O, allocation and release of memory, IPC, etc.

- Normal data processing in Oracle happens in “userland”, thus is not visible with strace.

2. strace

- The combination of strace + sql_trace is powerful, and gives more insight.
  - In our example, it shows some I/Os indeed are not instrumented.

- strace shows the result of the system call (!)
  - This means it shows the return code of the call.

strace - IOs with no event

```
io_submit(140425370206208, 1, {{0x7fb7516c45e8, 0, 0, 0, 257}}) = 1
io_getevents(140425370206208,2,128,{{0x7fb7516c4bc0,0x7fb7516c4bc0,1032192,0}}, {0, 0}) = 1
io_submit(140425370206208, 1, {{0x7fb7516c4bc0, 0, 0, 0, 257}}) = 1
io_getevents(140425370206208,2,128,{{0x7fb7516c45e8,0x7fb7516c45e8,1032192, 0}}, {0, 0}) = 1
io_submit(140425370206208, 1, {{0x7fb7516c45e8, 0, 0, 0, 257}}) = 1
io_getevents(140425370206208,1,128,{{0x7fb7516c4bc0,0x7fb7516c4bc0,1032192,0}}, {600,0}) = 1
write(8, "WAIT #140425323829824: nam='dire"..., 130) = 130
| 00000  57 41 49 54 20 23 31 34  30 34 32 35 33 32 33 38  WAIT #14 04253238 |
| 00010  32 39 38 32 34 3a 20 6e  61 6d 3d 27 64 69 72 65  29824: n am='dire |
| 00020  63 74 20 70 61 74 68 20  72 65 61 64 27 20 65 6c  ct path  read' el |
| 00030  61 3d 20 36 30 35 20 66  69 6c 65 20 6e 75 6d 62  a= 605 f ile numb |
| 00040  65 72 3d 35 20 66 69 72  73 74 20 64 62 61 3d 32  er=5 fir st dba=2 |
| 00050  35 34 37 34 20 62 6c 6f  63 6b 20 63 6e 74 3d 31  5474 blo ck cnt=1 |
| 00060  32 36 20 6f 62 6a 23 3d  32 30 30 34 35 20 74 69  26 obj#= 20045 ti |
| 00070  6d 3d 31 33 39 32 38 39  37 31 39 34 37 34 32 36  m=139289 71947426 |
| 00080  38 31                                            81                |
```

2. strace

- A lesser known ‘fe’ trace is it can leave out system calls.
  - This is how a s ace vel 8 plus strace looks like o **IO throtti** rocess

2. strace

- strace **does not leave out system calls**!
  - There is a shortcut/optimisation in libaio to prevent the call io_getevents from becoming a system call.
  - This is possible because the IO results are available in userspace.
  - Want to know more about this? See: http://fritshoogland.wordpress.com/2014/03/11/linux-strace-doesnt-lie-after-all/

2. strace

```
io_submit(140691720466432, 1, {{0x7ff55522de00, 0, 0, 0, 257}}) = 1
io_getevents(140691720466432, 1, 128, {{0x7ff55522e3d8, 0x7ff55522e3d8, 1032192, 0}}, {600, 0}) = 1
times({tms_utime=7, tms_stime=6, tms_cutime=0, tms_cstime=0}) = 430581853
write(8, "\n*** 2014-02-21 13:44:15.163\n", 29) = 29
 | 00000  0a 2a 2a 2a 20 32 30 31  34 2d 30 32 2d 32 31 20  .*** 201 4-02-21  |
 | 00010  31 33 3a 34 34 3a 31 35  2e 31 36 33 0a           13:44:15 .163.    |
lseek(8, 0, SEEK_CUR)                       = 5276
write(8, "WAIT #140691674130488: nam='dire"..., 134) = 134
 | 00000  57 41 49 54 20 23 31 34  30 36 39 31 36 37 34 31  WAIT #14 06916741 |
 | 00010  33 30 34 38 38 3a 20 6e  61 6d 3d 27 64 69 72 65  30488: n am='dire |
 | 00020  63 74 20 70 61 74 68 20  72 65 61 64 27 20 65 6c  ct path  read' el |
 | 00030  61 3d 20 31 39 39 37 39  37 36 20 66 69 6c 65 20  a= 19979 76 file  |
 | 00040  6e 75 6d 62 65 72 3d 35  20 66 69 72 73 74 20 64  number=5  first d |
 | 00050  62 61 3d 32 35 30 39 30  20 62 6c 6f 63 6b 20 63  ba=25090  block c |
 | 00060  6e 74 3d 31 32 36 20 6f  62 6a 23 3d 32 30 30 34  nt=126 o bj#=2004 |
 | 00070  35 20 74 69 6d 3d 31 33  39 32 39 38 36 36 35 35  5 tim=13 92986655 |
 | 00080  31 36 33 33 39 31                                 163391            |
```

2. strace

- This is how it looks like when the same is done using gdb:

2. strace

```
Breakpoint 5, 0x0000000007cfb4f2 in kslwtectx ()
Breakpoint 4, 0x0000000007cf47b6 in kslwtbctx ()
Breakpoint 5, 0x0000000007cfb4f2 in kslwtectx ()
Breakpoint 4, 0x0000000007cf47b6 in kslwtbctx ()
Breakpoint 5, 0x0000000007cfb4f2 in kslwtectx ()
Breakpoint 1, 0x00007f40a05c3660 in io_submit () from /lib64/libaio.so.1
Breakpoint 1, 0x00007f40a05c3660 in io_submit () from /lib64/libaio.so.1
Breakpoint 2, 0x000000000082d7d8 in io_getevents@plt ()
Breakpoint 2, 0x000000000082d7d8 in io_getevents@plt ()
Breakpoint 4, 0x0000000007cf47b6 in kslwtbctx ()
Breakpoint 2, 0x000000000082d7d8 in io_getevents@plt ()
Breakpoint 5, 0x0000000007cfb4f2 in kslwtectx ()
Breakpoint 1, 0x00007f40a05c3660 in io_submit () from /lib64/libaio.so.1
Breakpoint 2, 0x000000000082d7d8 in io_getevents@plt ()
Breakpoint 2, 0x000000000082d7d8 in io_getevents@plt ()
Breakpoint 4, 0x0000000007cf47b6 in kslwtbctx ()
Breakpoint 2, 0x000000000082d7d8 in io_getevents@plt ()
Breakpoint 5, 0x0000000007cfb4f2 in kslwtectx ()
```

2. strace

- With strace, we see:
  - single io_submit()
  - followed by a single io_getevents()

- With gdb, we see:
  - single io_submit()
  - followed by 3 io_getevents() calls.

2. strace

- IMPORTANT!
  - With strace, system calls are shown.
  - With write output, we see the Oracle wait events
  - When waits follow a system call or system calls, we can assume the wait is timing of that.

- There is no way to tell what the wait did actually time!!

2. strace

- This is the end of chapter 2.
  - In this chapter you have learned:
    - What strace does.
    - What limitation strace has (no user land).
    - That strace can’t tell you what a wait event actually did time.
    - ~~That it can leave out system calls.~~

3. gdb theory

- gdb is the GNU debugger.
  - http://www.sourceware.org/gdb/
  - Description from the website:

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

The program being debugged can be written in Ada, C, C++, Objective-C, Pascal (and many other languages). Those programs might be executing on the same machine as GDB (native) or on another machine (remote). GDB can run on most popular UNIX and Microsoft Windows variants.

3. gdb theory

- For profiling, we use the ‘make your program stop on specified conditions’.
  - However, we just use the stop notification, and make the process continue.

- To make full use of the gdb functionality, the executable needs to be compiled with the ‘-g’ flag to include debug information.

3. gdb theory

- By default, executables do not have debug information added.

- Debug information makes it able to relate C-code lines to steps of execution of the executable.

- As you might expect at this point: Oracle doesn't include this information.

3. gdb theory

- So…why talk about gdb and Oracle then?
  - Of course we still can use it, that is what this talk about.

- Oracle on linux is an ‘ELF’ executable.

```
[oracle@ol65vm [v11204] bin]$ file oracle
oracle: setuid setgid ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.18, not stripped
```

3. gdb theory

- It says ‘not stripped’
  - Which means function symbolic information is NOT removed.

- It is dynamically linked!
  - An executable has to symbolically call functions in a library.

3. gdb theory

- Check on dynamically loaded libraries with ‘ldd’:

```
[oracle@ol65vm [v11204] bin]$ ldd oracle
    linux-vdso.so.1 =>  (0x00007fffc0dfe000)
    libodm11.so => /u01/app/oracle/product/11.2.0.4/dbhome_1/lib/libodm11.so
(0x00007fdb6d75e000)
    libcell11.so => /u01/app/oracle/product/11.2.0.4/dbhome_1/lib/libcell11.so
(0x00007fdb6d5fa000)
    libskgxp11.so => /u01/app/oracle/product/11.2.0.4/dbhome_1/lib/libskgxp11.so
(0x00007fdb6d424000)
    libaio.so.1 => /lib64/libaio.so.1 (0x00007fdb6bcea000)
    libdl.so.2 => /lib64/libdl.so.2 (0x00007fdb6bae5000)
    libm.so.6 => /lib64/libm.so.6 (0x00007fdb6b861000)
    libpthread.so.0 => /lib64/libpthread.so.0 (0x00007fdb6b644000)
    libnsl.so.1 => /lib64/libnsl.so.1 (0x00007fdb6b42a000)
    libc.so.6 => /lib64/libc.so.6 (0x00007fdb6b096000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fdb6d861000)
(partial listing)
```

3. gdb theory

- In order to call a function in a library, it needs to be defined somewhere in the executable:
  - Procedure Linkage Table (plt).
  - Libraries and functions are lazy loaded.

```
[oracle@ol65vm [v11204] bin]$ readelf -a oracle | grep io_getevents
00000b5de698  00fc00000007 R_X86_64_JUMP_SLO 0000000000000000 io_getevents + 0
   252: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND io_getevents@LIBAIO_0.4 (11)
633326: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND io_getevents@@LIBAIO_0.4
```

3. gdb theory

- The reason for showing the plt:
  - The functions in a dynamically linked executable can only be called by name.
  - So the library is free to have the function at any offset.

3. gdb theory

- Functions in an executable can be found using ‘nm’, if it’s not stripped:

```
[oracle@ol65vm [v11204] bin]$ nm -a oracle | grep kslwtbctx$
0000000008c1867c T kslwtbctx
```

3. gdb theory

- Again: I don’t suspect Oracle to ever release the debug symbols for the Oracle executable.
  - Because it would reveal source code.
  - This also means that gdb does not understand the function call arguments.

- But Linux is open source!

3. gdb theory

- Oracle linux has a complete repository with the operating system and utilities compiled with debug symbols: the debug info packages.
- https://oss.oracle.com/ol6/debuginfo

- For the sake of this course, I created a directory ~root/debuginfo_rpms in which the relevant debuginfo rpms are put.

3. gdb theory

- Install the debug info packages from that directory:

```
[oracle@ol65vm [] ~]$ su -
Password:
[root@ol65vm ~]# cd ~/debuginfo_rpms/
[root@ol65vm debuginfo_rpms]# yum localinstall *
Setting up Local Package Process
Examining glibc-debuginfo-2.12-1.132.el6.x86_64.rpm: glibc-
debuginfo-2.12-1.132.el6.x86_64
…
Total size: 54 M
Installed size: 54 M
Is this ok [y/N]: y
…
```

3. gdb theory

- Verify the installed debug info packages:

```
[root@ol65vm debuginfo_rpms]# rpm -qa | grep debuginfo

glibc-debuginfo-common-2.12-1.132.el6.x86_64
libaio-debuginfo-0.3.107-10.el6.x86_64
glibc-debuginfo-2.12-1.132.el6.x86_64
```

3. gdb theory

- Now let's setup a normal database connection again:

```
[oracle@ol65vm [v11204] bin]$ rs ts/ts@//localhost/v11204
```

- Find the PID of the server process in a root session, and attach to it with gdb again:

```
[root@ol6vm ~]# ps -ef | grep [L]OCAL=NO
oracle    2657     1  0 02:22 ?        00:00:00 oraclev11204 (LOCAL=NO)
[root@ol6vm ~]# gdb -p 2657
```

3. gdb theory

- Enter the following in the gdb session:

```
set pagination off
break io_submit
break io_getevents_0_4
c
```

- This makes the sqlplus session runnable again.

```
select count(*) from t2;
```

3. gdb theory

- Because we didn't specify continue as action with a break, gdb breaks execution.
- Look what gdb tells you with io_submit():

```
Breakpoint 1, io_submit (ctx=0x7ff8b626c000, nr=1, iocbs=0x7fffa5c31a80) at
io_submit.c:23
23 io_syscall3(int, io_submit, io_submit, io_context_t, ctx, long, nr,
struct iocb **, iocbs)
```

- continue until you get a io_getevents():

```
Breakpoint 2, io_getevents_0_4 (ctx=0x7ff8b626c000, min_nr=2, nr=128,
events=0x7fffa5c37b68, timeout=0x7fffa5c38b70) at io_getevents.c:46
46    if (ring==NULL || ring->magic != AIO_RING_MAGIC)
```

3. gdb theory

- Using the debug info package, gdb can interpret system calls arguments for which the debug information has been installed!
  - This allows you to see quite a lot.

- Important: gdb breaks if the function is *entered*. This means we can not read the return code of the function we are breaking on!
  - io_getevents() *returns* ready IOs.

3. gdb theory

- Quit the gdb (root) session (q).
- Quit the sqlplus (oracle) session.
- In the oracle session:

```
cd ~/gdb_macros
sq
@create_pw
@create_pl
exit
```

3. gdb theory

- Now setup a normal oracle connection again:

```
rs ts/ts@//localhost/v11204
```

- And setup a gdb session from ~oracle/ gdb_macros and attach to the oracle session:

```
cd ~oracle/gdb_macros
ps -ef | grep [L]OCAL=NO
gdb -p PID
```

3. gdb theory

- Now use some already prepared gdb macros:

```
source pstruct.gdb
source paio.gdb
source pw.gdb
c
```

- And start ‘select count(*) from t2;’ from the sqlplus session.

3. gdb theory

- Now look what is visible with the three gdb macros:

```
kslwtectx -- Previous wait time: 74621038: SQL*Net message from client
opiosq0 -- parse
kksfbc -- V$SQL.ADDRESS: 78d7a6d8
opiexe -- execute
kslwtbctx
kslwtectx -- Previous wait time: 1195: SQL*Net message to client
opifch2 -- fetch
kslwtbctx
io_submit - 1,58f5c000 - nr,ctx
 fd: 256, nbytes: 1
io_getevents - 1,58f5c000 - minnr,ctx,timeout: $1 = {tv_sec = 10, tv_nsec = 0}
kslwtectx -- Previous wait time: 70121: Disk file operations I/O
```

3. gdb theory

- This explains the direct path read waits:
  - This is done with cgroups IO throttling

```
opifch2 -- fetch
io_submit - 1,58f5c000 - nr,ctx
 fd: 256, nbytes: 106496
io_submit - 1,58f5c000 - nr,ctx
 fd: 256, nbytes: 122880
io_getevents - 2,58f5c000 - minnr,ctx,timeout: $1 = {tv_sec = 0, tv_nsec = 0}
io_getevents - 2,58f5c000 - minnr,ctx,timeout: $2 = {tv_sec = 0, tv_nsec = 0}
kslwtbctx
io_getevents - 1,58f5c000 - minnr,ctx,timeout: $3 = {tv_sec = 600, tv_nsec = 0}
kslwtectx -- Previous wait time: 946495: direct path read
```

3. gdb theory

- This is the end of chapter 3.
  - In this chapter you have learned:
    - What gdb is.
    - What debug symbols are and how to get them.
    - What stripped and not stripped means.
    - oracle as dynamically linked executable.
    - procedure linkage table and finding functions
    - how to break on functions

3. gdb theory

- The breaking in gdb happens on entering a function call.
- The existence of gdb macro's created by Enkitec to obtain more information.

4. gdb lab

- What does the wait event ‘direct path read’ mean
  - Enable synchronous IO for the database now.
    - filesystemio_options=‘none’
- Now trace the ‘select count(*) from t2’ again with gdb.
- What wait event indicates IO now? What does the wait event time?
- Turn on asynchronous IO again when finished.

4. gdb lab

- What does asynchronous IO mean for IO calls?
  - Synchronous IO is done with the pread() call.
  - Asynchronous IO is done with the calls io_submit() and io_getevents().
- Flush the buffer cache.
- Trace the ‘select count(*) from t2’ with gdb.
- What IO calls are done?

4. gdb lab

- What does SIO mean for the dbwr wait events?
  - Enable synchronous IO for the database now.
    - filesystemio_options='none'
  - The dbwr has a distinct write wait event.
  - Trace the dbwr (set db_writer_processes to 1 if there are multiple) with gdb.
  - Create a table, insert a row, commit and issue 'alter system checkpoint'.
  - Turn on asynchronous IO again when finished.

5. Extra

- How to find function names?
  – perf
- Show how some of the gdb macro's are made.
- Explain using registers to find function call arguments.