# 最老的新技术：调试Oracle技术实战
## ----在堆栈中寻找异常宕库原因

ebay 首要数据库工程师 吕海波（VAGE）

DTCC

2015中国数据库技术大会

DATABASE TECHNOLOGY CONFERENCE CHINA 2015

大数据技术探索和价值发现

IT168 ChinaUnix ITPUB

# 内容简介

内容简介：

- 什么是调试Oracle技术
- 程序的机器级表示
- 断点
- 发现断点
- 神奇的等待事件：等待事件原理分析
- 案例

# 什么是调试Oracle技术

调试Oracle就是对Oracle进行逆向工程，再通俗点说，就是反汇编Oracle。

相信听到这个结果的朋友，一定会认为：

# 什么是调试Oracle技术

调试Oracle就是对Oracle进行逆向工程，再通俗点说，就是反汇编Oracle。

相信听到这个结果的朋友，一定会认为：

# 什么是调试Oracle技术

但其实不是这样的， Oracle的逆向工程没有你想像中的哪么难。它是真正的纸老虎。

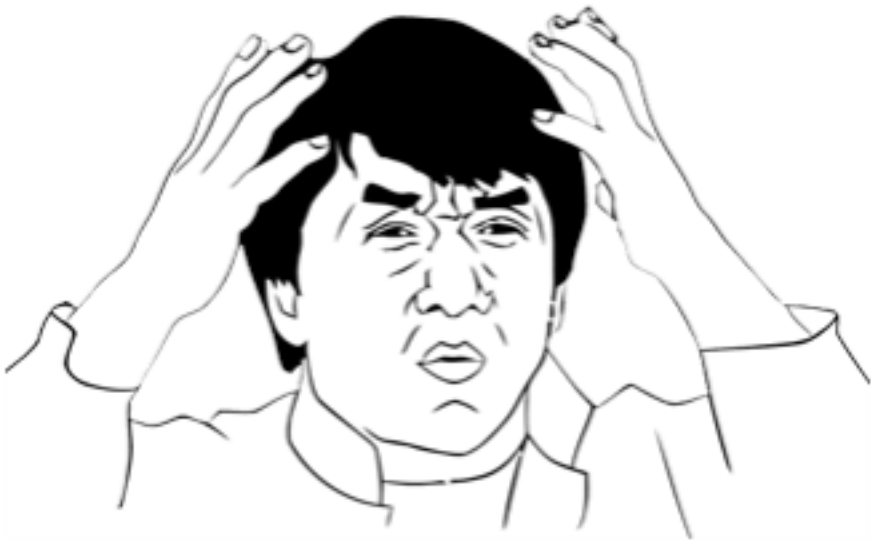
# 什么是调试Oracle技术：逆向工程

我们的目的，并不是要读懂Oracle每一条反汇编代码，我相信这不可能，代码量太大了。我们只需要从反编代码找出我们感兴趣的片断即可。这样一来，难度就大大下降，再借助现在优秀的调试工具，gdb/mdb和DTrace，对于有开发功底的DBA来说，可以说易如反掌。

# 调试Oracle技术有什么用

在开始之前，还要回答一个问题：

为什么要这么做？或者，调试Oracle有什么用。



搞这么难的东西有什么用？

# 调试Oracle技术有什么用

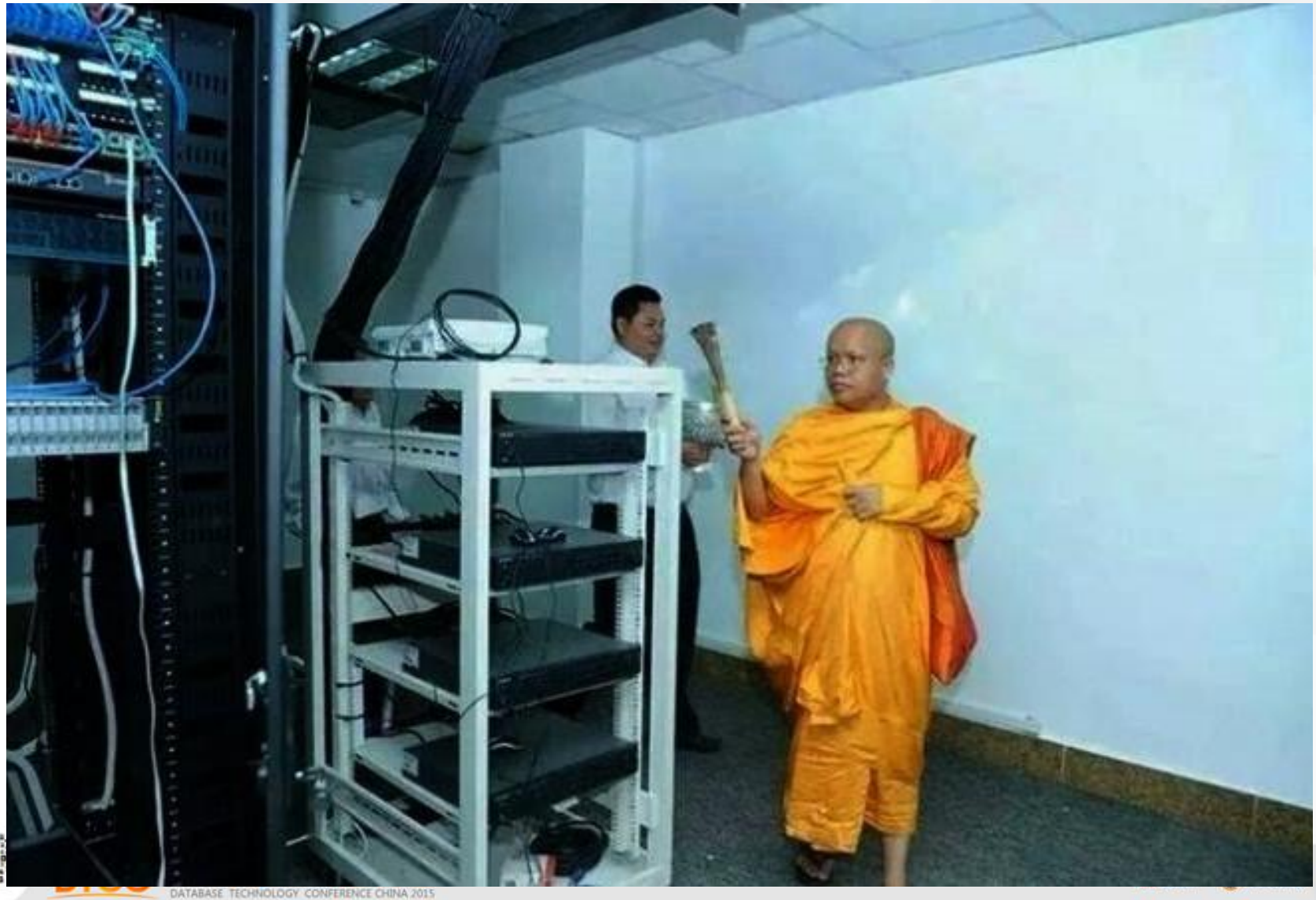
答案很简单，一是这东西其实没相像中哪么难。二是在我做DBA的职业生涯中，总有些疑难问题，让我有求神拜佛的冲动。

# 什么是调试Oracle技术

就像这样：

# 调试Oracle技术有什么用

或这样：



# 什么是调试Oracle技术

告诉浙江地区DBA一个经验，

我2010年初，去灵隐上完香之后，大半年内，事故率没有下降

2010年底去普陀山上完香之后，大半年内事故率下降了50%

# 什么是调试Oracle技术

除了去普陀山上香外，我想，我也要做点什么。不能老是麻烦神仙帮忙。

于是，我就开始了我的“调试Oracle”之旅。

# 程序的机器级表示

让我们从一个简单的小例子开始吧。

有个笑话，程序员准备练书法，提起狼毫笔，占足墨汁，在徽州宣纸上写下饱满的

Holle word

# 程序的机器级表示

很可惜，我们不能Hello Word开始，它包含的元素太少了，我们从下面这个例子开始。

它包含几个对我们来说最基本的元素：变量（局部变量、全局变量）、函数调用、参数传递。

```
int result;

void add(int p1, int p2)
{
    result=p1+p2;
    return ;
}

int main()
{
    int a,b;
    a=1;
    b=2;
    add(a,b);
    printf("%d+%d=%d\n",a,b,result);
    return 0;
}
```

这个程序很简单，先了解一下它是干吗的。然后，我们就先从它开始。

# 程序的机器级表示

它包含几个对我们来说最基本的元素：变量（局部变量、全局变量）、函数调用、参数传递。

```
int result;

void add(int p1, int p2)
{
    result=p1+p2;
    return ;
}

int main()
{
    int a,b;
    a=1;
    b=2;
    add(a,b);
    printf("%d+%d=%d\n",a,b,result);
    return 0;
}
```

```
main:          pushq  %rbp
main+1:        movq   %rsp,%rbp
main+4:        subq   $0x10,%rsp
main+8:        movl   $0x1,-0x4(%rbp)
main+0xf:      movl   $0x2,-0x8(%rbp)
main+0x16:     movl   -0x8(%rbp),%esi
main+0x19:     movl   -0x4(%rbp),%edi
main+0x1c:     call   -0x34    <add>
main+0x21:     movl   -0x8(%rbp),%edx
main+0x24:     movl   -0x4(%rbp),%esi
main+0x27:     movl   0x103e3(%rip),%ecx
main+0x2d:     movl   $0x400df0,%edi
main+0x32:     movl   $0x0,%eax
main+0x37:     call   -0x19f   <PLT:printf>
main+0x3c:     movl   $0x0,%eax
main+0x41:     leave
main+0x42:     ret
```

右边是main函数的反汇编代码，我们逐条对照的了解一下它的意义，你就会发现，其实这很简单。

# 程序的机器级表示

“xx(%rbp)” 这样形式的，括号中是rbp的，都是局部变量。

```
int result;

void add(int p1, int p2)
{
    result=p1+p2;
    return ;
}

int main()
{
    int a,b;
    a=1;
    b=2;
    add(a,b);
    printf("%d+%d=%dWn",a,b,result);
    return 0;
}
```

```
main:       pushq  %rbp
main+1:     movq   %rsp,%rbp
main+4:     subq   $0x10,%rsp
main+8:     movl   $0x1,-0x4(%rbp)   // a=1
main+0xf:   movl   $0x2,-0x8(%rbp)   // b=2
main+0x16:  movl   -0x8(%rbp),%esi
main+0x19:  movl   -0x4(%rbp),%edi
main+0x1c:  call   -0x34    <add>
main+0x21:  movl   -0x8(%rbp),%edx
main+0x24:  movl   -0x4(%rbp),%esi
main+0x27:  movl   0x103e3(%rip),%ecx
main+0x2d:  movl   $0x400df0,%edi
main+0x32:  movl   $0x0,%eax
main+0x37:  call   -0x19f   <PLT:printf>
main+0x3c:  movl   $0x0,%eax
main+0x41:  leave
main+0x42:  ret
```

另外，补充一点基础知识，xx(%rbp)，这种方式叫“基址加偏移量寻址”。rbp中保存一个内存地址，也就是“基址”，此基址加上偏移量xx，就是我们这里要操作的目标内存。打个比方，“天安门广场向东100米”，等于：100米（天安门广场）。基址是“天安门广场”，偏移量是100米。

# 程序的机器级表示

```
int result;

void add(int p1, int p2)
{
    result=p1+p2;
    return ;
}

int main()
{
    int a,b;
    a=1;
    b=2;
    add(a,b);
    printf("%d+%d=%dWn",a,b,result);
    return 0;
}
```

```
main:       pushq  %rbp
main+1:     movq   %rsp,%rbp
main+4:     subq   $0x10,%rsp
main+8:     movl   $0x1,-0x4(%rbp)   // a=1
main+0xf:   movl   $0x2,-0x8(%rbp)   // b=2
main+0x16:  movl   -0x8(%rbp),%esi
main+0x19:  movl   -0x4(%rbp),%edi
main+0x1c:  call   -0x34    <add>
main+0x21:  movl   -0x8(%rbp),%edx
main+0x24:  movl   -0x4(%rbp),%esi
main+0x27:  movl   0x103e3(%rip),%ecx
main+0x2d:  movl   $0x400df0,%edi
main+0x32:  movl   $0x0,%eax
main+0x37:  call   -0x19f   <PLT:printf>
main+0x3c:  movl   $0x0,%eax
main+0x41:  leave
main+0x42:  ret
```

将变量a,b传到esi和edi中作为参数，为调用add函数作准备

调用add函数

注意这里，64位系统中，调用函数传递参数时，第一个参数只能放在rdi中，第二个参数只能在 rsi中，第三、第四等等参数，只能依次放入： rdx, rcx, rbx, r8, r9, r10, ……等寄存器中。

# 程序的机器级表示

用以下两步完成一个加法：
Eax=p1;
Eax=eax+p2;

```
int result;

void add(int p1, int p2)
{
    result=p1+p2;
    return ;
}

int main()
{
    int a,b;
    a=1;
    b=2;
    add(a,b);
    printf("%d+%d=%dWn",a,b,r
    return 0;
}
```

```
add:        pushq   %rbp
add+1:      movq    %rsp,%rbp
add+4:      movl    %edi,-0x4(%rbp) // p1=1
add+7:      movl    %esi,-0x8(%rbp) // p2=2
add+0xa:    movl    -0x8(%rbp),%eax // eax=p1
add+0xd:    addl    -0x4(%rbp),%eax // eax=eax+p2
add+0x10:   movl    %eax,0x1041(%rip)  // result=eax
add+0x16:   leave
add+0x17:   ret
```

“-0x4(%rbp)”和“-0x8(%rbp)”是局部变量。Edi和esi中是调用者要传递的第一、二个参数。这两步的意义，是将参数值传递到变量p1和p2中。

“xxx(%rip)”这样形式的东西，代表是全局变量

# 程序的机器级表示

函数的参数，会依次放入rdi, rsi, rdx, rcx, rbx, r8, r9, r10, ……等寄存器

```
int result;

void add(int p1, int p2)
{
    result=p1+p2;
    return ;
}

int main()
{
    int a,b;
    a=1;
    b=2;
    add(a,b);
    printf("%d+%d=%d\n",a,b,result);
    return 0;
}
```

```
main:       pushq  %rbp
main+1:     movq   %rsp,%rbp
main+4:     subq   $0x10,%rsp
main+8:     movl   $0x1,-0x4(%rbp)  // a=1
main+0xf:   movl   $0x2,-0x8(%rbp)  // b=2
main+0x16:  movl   -0x8(%rbp),%esi
main+0x19:  movl   -0x4(%rbp),%edi
main+0x1c:  call   -0x34    <add>
main+0x21:  movl   -0x8(%rbp),%edx      //第三个参数,变量b
main+0x24:  movl   -0x4(%rbp),%esi      //第二个参数,变量a
main+0x27:  movl   0x103e3(%rip),%ecx   //第四个参数,result
main+0x2d:  movl   $0x400df0,%edi       //第一个参数,格式字符
串
main+0x32:  movl   $0x0,%eax
main+0x37:  call   -0x19f   <PLT:printf> //调用printf
main+0x3c:  movl   $0x0,%eax
main+0x41:  leave
main+0x42:  ret
```

# 程序的机器级表示

```
int result;

void add(int p1, int p2)
{
    result=p1+p2;
    return ;
}

int main()
{
    int a,b;
    a=1;
    b=2;
    add(a,b);
    printf("%d+%d=%d\n",a,b,result);
    return 0;
}
```

```
main:       pushq  %rbp
main+1:     movq   %rsp,%rbp
main+4:     subq   $0x10,%rsp
main+8:     movl   $0x1,-0x4(%rbp)  // a=1
main+0xf:   movl   $0x2,-0x8(%rbp)  // b=2
main+0x16:  movl   -0x8(%rbp),%esi
main+0x19:  movl   -0x4(%rbp),%edi
main+0x1c:  call   -0x34    <add>
main+0x21:  movl   -0x8(%rbp),%edx      //第三个参数,变量b
main+0x24:  movl   -0x4(%rbp),%esi      //第二个参数,变量a
main+0x27:  movl   0x103e3(%rip),%ecx   //第四个参数,result
main+0x2d:  movl   $0x400df0,%edi       //第一个参数,格式字符串
main+0x32:  movl   $0x0,%eax
main+0x37:  call   -0x19f   <PLT:printf> //调用printf
main+0x3c:  movl   $0x0,%eax
main+0x41:  leave
main+0x42:  ret
```

函数的返回值放入eax寄存器。

# 程序的机器级表示

阅读反汇编代码，难吗？只是传说中很难。所以说，一切帝国主义，都是：

# 程序的机器级表示

更多信息，请参考《深入理解计算机系统》第三章，此章的名字就是“程序的机器级表示”。

在此章中，有详细的C语言的条件、分支和各种循环和汇编的对应关系。读完这章，你会发现，阅读Oracle反汇编的片段，是很简单的事。

有人经常和我聊：“你花了这么多时间去研究这个，值得吗”。

先不说值不值的问题，其实，这么简单的东西，真的花不了太多时间。

# 程序的机器级表示

有了基本的汇编基础，哪么，理解后面的问题也就简单了。下面我们讨论下

一个问题：**断点**。

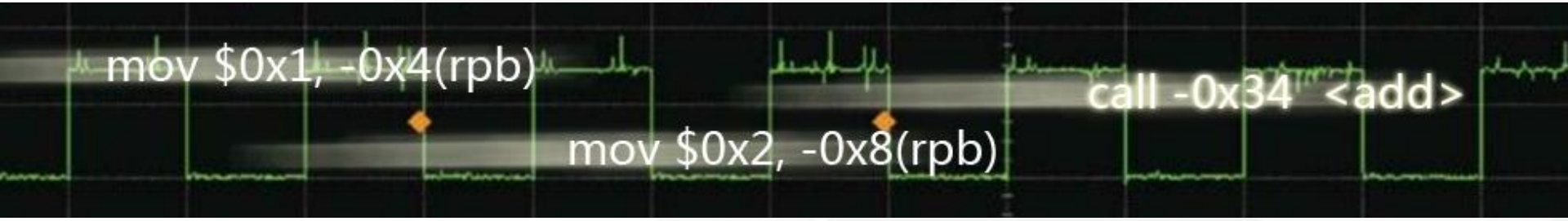
断点，是程序调试的重要工具。断点就是让程序在某个地方停下来，然后我们可以慢慢观察程序的状态。

设置断点，需要使用调试工具：gdb/mdb。断点命令是：

- mdb: 函数名:b
- gdb : b 函数名

# 断点

程序，对CPU来说，不过一串长长的指令流:



# 断点

CPU逐条的执行命令。

。

```
main:          pushq  %rbp
main+1:        movq   %rsp,%rbp
main+4:        subq   $0x10,%rsp
main+8:        movl   $0x1,-0x4(%rbp)  // a=1
main+0xf:      movl   $0x2,-0x8(%rbp)  // b=2
main+0x16:     movl   -0x8(%rbp),%esi
main+0x19:     movl   -0x4(%rbp),%edi
main+0x1c:     call   -0x34    <add>
main+0x21:     movl   -0x8(%rbp),%edx
main+0x24:     movl   -0x4(%rbp),%esi
main+0x27:     movl   0x103e3(%rip),%ecx
main+0x2d:     movl   $0x400df0,%edi
main+0x32:     movl   $0x0,%eax
main+0x37:     call   -0x19f   <PLT:printf>
main+0x3c:     movl   $0x0,%eax
main+0x41:     leave
main+0x42:     ret

add:           pushq  %rbp
add+1:         movq   %rsp,%rbp
add+4:         movl   %edi,-0x4(%rbp)
add+7:         movl   %esi,-0x8(%rbp)
add+0xa:       movl   -0x8(%rbp),%eax
add+0xd:       addl   -0x4(%rbp),%eax
add+0x10:      movl   %eax,0x10412(%rip)
add+0x16:      leave
add+0x17:      ret
```

# 断点

如果你想让CPU执行到某个命令处暂停，那么，就把哪条命令设置为断点

比如，我想让CPU停在黑底白字处的位置。

这里是add函数的入口点，我们可以使用mdb命令”**add:b**”，在此处设置断点。

```
main:          pushq  %rbp
main+1:        movq   %rsp,%rbp
main+4:        subq   $0x10,%rsp
main+8:        movl   $0x1,-0x4(%rbp)   // a=1
main+0xf:      movl   $0x2,-0x8(%rbp)   // b=2
main+0x16:     movl   -0x8(%rbp),%esi
main+0x19:     movl   -0x4(%rbp),%edi
main+0x1c:     call   -0x34     <add>
main+0x21:     movl   -0x8(%rbp),%edx
main+0x24:     movl   -0x4(%rbp),%esi
main+0x27:     movl   0x103e3(%rip),%ecx
main+0x2d:     movl   $0x400df0,%edi
main+0x32:     movl   $0x0,%eax
main+0x37:     call   -0x19f    <PLT:printf>
main+0x3c:     movl   $0x0,%eax
main+0x41:     leave
main+0x42:     ret

add:           pushq  %rbp
add+1:         movq   %rsp,%rbp
add+4:         movl   %edi,-0x4(%rbp)
add+7:         movl   %esi,-0x8(%rbp)
add+0xa:       movl   -0x8(%rbp),%eax
add+0xd:       addl   -0x4(%rbp),%eax
add+0x10:      movl   %eax,0x10412(%rip)
add+0x16:      leave
add+0x17:      ret
```

# 断点

CPU在执行到黑底白字的指令处，将会停止，这里就是add函数的入口处。

```
main:          pushq  %rbp
main+1:        movq   %rsp,%rbp
main+4:        subq   $0x10,%rsp
main+8:        movl   $0x1,-0x4(%rbp)   // a=1
main+0xf:      movl   $0x2,-0x8(%rbp)   // b=2
main+0x16:     movl   -0x8(%rbp),%esi
main+0x19:     movl   -0x4(%rbp),%edi
main+0x1c:     call   -0x34     <add>
main+0x21:     movl   -0x8(%rbp),%edx
main+0x24:     movl   -0x4(%rbp),%esi
main+0x27:     movl   0x103e3(%rip),%ecx
main+0x2d:     movl   $0x400df0,%edi
main+0x32:     movl   $0x0,%eax
main+0x37:     call   -0x19f    <PLT:printf>
main+0x3c:     movl   $0x0,%eax
main+0x41:     leave
main+0x42:     ret
```

这里是断点 →

```
add:           pushq  %rbp
add+1:         movq   %rsp,%rbp
add+4:         movl   %edi,-0x4(%rbp)
add+7:         movl   %esi,-0x8(%rbp)
add+0xa:       movl   -0x8(%rbp),%eax
add+0xd:       addl   -0x4(%rbp),%eax
add+0x10:      movl   %eax,0x10412(%rip)
add+0x16:      leave
add+0x17:      ret
```

# 断点

对调试Oracle来说，断点作用巨大。在后面马上就有一个例子，演示断点的作用。

我们自己写的程序，我可以知道程序中有个子函数：add，我们可以在add处设置断点。
对于Oracle，它完全是一个黑盒子，我们又要在哪里设置断点呢？

# 发现断点

是DTrace登场的时候了。

DTrace发源于Solaris系统，虽然现在也移植到Linux下，但Linux下的DTrace确少一个重要功能，目前我们还不能使用它来“发现断点”。

注意：

虽然DTrace只能在Solaris下，但Oracle在所有OS系统中的原理基本都是一致的，因此我们发现的Oracle规则、原理，可以用于所有Oracle系统。

# 发现断点

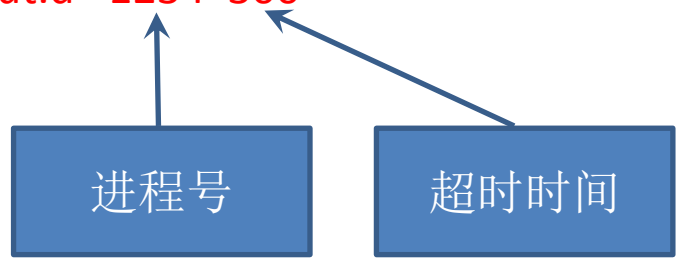
DTrace内置了很多探针，用DBA的语言来说，相当于Solaris系统中内置了很多触发器，这些触发器平常是Disable的，你可以使用DTrace enable指定的触发器。同时，你还可以为这个触发器定义动作，也就是当此触发器被触发时要作什么。通常“动作”就是使用printf函数显示一些参数或内存的值。

当然，除了显示内存值，也可以修改。比如，想研究一下Oracle DBWR进程写脏块有没有什么调优空间，哪么第一步就是要先了解 DBWR进程的工作原理，DBWR有个3秒超时机制，但除DBWR外，Oracle很多进程都有3秒超时机制，哪么多进程大家混在一起超时，也不知道是谁的超时影响了系统，我在我的书《Oracle内核技术揭密》中，有一段脚本，可以修改进程的超时时间。在这儿，把这段脚本分享一下：

# 发现断点

修改进程超时时间：

```
bash-3.2# cat time_out.d
#!/usr/sbin/dtrace -s -n

struct timespec *timeout;

pid$1::semtimedop:entry
{
        timeout=(struct timespec *)copyin(arg3,8);
        timeout->tv_sec=$2;
        copyout(timeout,arg3,8);
}

Bash-3.2# time_out.d   1234  300
```




# 发现断点

更多DTrace相关的基础知识，可以参考我在ITPUB上的帖子。

使用DTrace可以做什么呢？
它可以把Oracle的函数名都显示出来。这是我们“调试Oracle”的第一步，也是我们打开Oracle这个黑盒子的第一步。

只需要用下面短短几行代码，就可以得到Oracle在进行某个操作时的所调用的函数名，以及函数相关参数：

```
#!/usr/sbin/dtrace -s -n

dtrace:::BEGIN
{
	i=1;
}
pid$1:::entry
{
	printf("i=%d %s(%x,%x,%x,%x,%x,%x);",i, probefunc,arg0,arg1,arg2,arg3,arg4,arg5);
	i=i+1;
}
```

# 发现断点

这是跟踪下列测试SQL执行的结果：

```
SQL> select * from vage where rowid='AAADLMAAEAAAACDAAA';
        ID NAME
---------- --------------------
         1 aaaaaa
```

```
bash-3.2# chmod 755 get_func.d
bash-3.2# ./get_func.d 1340
dtrace: script './get_func.d' matched 152978 probes
CPU     ID                    FUNCTION:NAME
  0 201201                      memcpy:entry i=1 memcpy(fffffd7fffdfc8eb,d4bd860,1,1,d4bd9a2,142);
  0  52479                  kslwtectx:entry i=2 kslwtectx(fffffd7fffdfc610,d4bd860,d4bd861,fffffd7fffdfc8ec,ced87ac,11);
  0 203504                  gethrtime:entry i=3 gethrtime(fffffd7fffdfc610,d4bd860,d4bd861,fffffd7fffdfc8ec,0,11);
  0  52554         kslwt_end_snapshot:entry i=4 kslwt_end_snapshot(395625898,395625898,1,1ee029adacf7,ced881c,ced87a8);
  0  72715      kews_update_wait_time:entry i=5 kews_update_wait_time(6,20735b7,1f,159,395ed9f60,afe770c);
  0  52268                  kskthewt:entry i=6 kskthewt(ef5b7e59,0,6,0,0,394d0f7e0);
  0 201201                      memcpy:entry i=7 memcpy(fffffd7fffdfc8f6,d4bd861,2,2010,d4bd863,395623bb0);
  0 120954                   kpuhhmrk:entry i=8 kpuhhmrk(cedd8e0,d4bd861,d4bd863,11,ced8c68,1068);
  0 135665                  kpggGetPG:entry i=9 kpggGetPG(cedd8e0,d4bd861,d4bd863,11,fffffd7ffcb21f90,fffffd7ffcb21f90);
  0 201208                     setjmp:entry i=10 setjmp(fffffd7fffdfc758,d4bd861,d4bd863,11,ceed7f8,bc794f8);
  0 124896                     kghmrk:entry i=11 kghmrk(ceed648,fffffd7ffcb21000,0,1,cedd8e0,fffffd7fffdfc730);
  0 123372                     ttcpip:entry i=12 ttcpip(cedd950,69,fffffd7fffdfc920,0,fffffd7fffdfdaa0,fffffd7fffdfc8fc);
  0 135665                  kpggGetPG:entry i=13 kpggGetPG(cedd950,69,fffffd7fffdfc920,0,fffffd7ffcb21f90,fffffd7fffdfc8fc);
  0 201201                      memcpy:entry i=14 memcpy(fffffd7fffdfc920,d4bd863,10,0,d4bd873,1);
  0 201201                      memcpy:entry i=15 memcpy(fffffd7fffdfc7b0,d4bd873,4,0,d4bd877,1);
  0  77515                     opiodr:entry i=16 opiodr(69,2,fffffd7fffdfc920,0,41e3a80,bc76740);
  0 201202                     memset:entry i=17 memset(fffffd7ffcb4bc58,0,28,0,ced8c30,ced9a70);
  0 201208                     setjmp:entry i=18 setjmp(fffffd7fffdfbec8,0,0,0,bc794f8,ced9a70);
  0  53523                    ksupucg:entry i=19 ksupucg(1,c7267b8,28d,1,380021018,fffffd7fffdfbea0);
  0 127726                     slcpu:entry i=20 slcpu(ceed278,c7267b8,28d,8,ceed2b0,391712210);
  0 203596                     times:entry i=21 times(fffffd7fffdfbaf0,c7267b8,28d,8,ceed2b0,391712210);
  0 203504                  gethrtime:entry i=22 gethrtime(fffffd7fffdfbaf0,c7267b8,0,fffffd7ffcd92caa,2008,64);
  0  52413       ksl_get_shared_latch:entry i=23 ksl_get_shared_latch(395683950,1,395623bb0,56,8,1);
  0 128134                  skgslocas:entry i=24 skgslocas(395683950,0,1,fffffd7fffdfbac0,0,bc7acc8);
  0  52418                     kslfre:entry i=25 kslfre(395683950,1,2,1,38001bc20,ceed648);
  0 128136                 sskgsldecr:entry i=26 sskgsldecr(395683950,1,ffffffff,10,395683950,e);
  0  59601                   ktcsptg:entry i=27 ktcsptg(39567a4e0,0,0,10,0,ced8cc0);
  0 103289                   kscdnfy:entry i=28 kscdnfy(1,39567a450,0,10,38000af10,395623bb0);
  0  60477               kticallpush:entry i=29 kticallpush(1,39567a450,ff,ff,1,fffffd7ffc9df470);
  0  53171            ksptch_callpush:entry i=30 ksptch_callpush(1,39567a450,ff,ff,fffffd7ffc9df480,0);
```

看到满屏奇奇怪怪的函数名，很多人在这一步放弃了。其实，这才刚刚开始。理解它们很简单。下面我们就以等待事件为例，说一下如何从这里面发现价值。

# 神奇的等待事件

有没有觉得Oracle的等待事件非常神奇？它是我们DBA的重要工具。它的原理是什么呢？下面，我们就以它为例，使用“调试Oracle”技术，详细分析等待事件的原理。

# 神奇的等待事件

万事开头难。为了研究等待事件，我还是花了点时间开头的。DTrace中有一个简单的方法，可以统计调用每个函数的次数。我就是从这个次数开始的。

执行测试SQL：“select * from vage where rowid='AAADLMAAEAAAACDAAA'”，当是软软解析、逻辑读时，在没有竞争的情况下，会有四次等待事件：

两次SQL*Net message to client
两次SQL*Net message from client

关于这点，可以很容易的从v$session_event中得到。

# 神奇的等待事件

然后，我用如下脚步跟踪测试SQL的执行：

```
#!/usr/sbin/dtrace -s -n

dtrace:::BEGIN
{
      printf("Start...\n");
}


pid$1:::entry
{
      @counts[probefunc]=count();
}

dtrace:::END
{
      trace("--------------------------");
      printa(@counts);
}
```

关键在于这里，它统计所有函数的调用次数

# 神奇的等待事件

因为测试SQL一共会有4次等待事件，所以我只观注调用次数为4的函数，这些函数共有15个：

| | |
|---|---|
| KGHISPIR | 4 |
| _save_nv_regs | 4 |
| kews_sqlcol_begin | 4 |
| kews_update_wait_time | 4 |
| kghxhal | 4 |
| kghxhfr | 4 |
| kglGetMutex | 4 |
| kksGetStats | 4 |
| kskthbwt | 4 |
| kskthewt | 4 |
| kslwt_end_snapshot | 4 |
| kslwt_start_snapshot | 4 |
| kslwtbctx | 4 |
| kslwtectx | 4 |
| opikndf2 | 4 |

在这些函数里面，一定有一些是关于等待事件的函数？

# 神奇的等待事件

经过观察，如下几个函数引起我的注意，原因很简单，它们的名字中带有”wt”：

| | |
|---|---|
| KGHISPIR | 4 |
| _save_nv_regs | 4 |
| kews_sqlcol_begin | 4 |
| kews_update_wait_time | 4 |
| kghxhal | 4 |
| kghxhfr | 4 |
| kglGetMutex | 4 |
| kksGetStats | 4 |
| **kskthbwt** | **4** |
| **kskthewt** | **4** |
| **kslwt_end_snapshot** | **4** |
| **kslwt_start_snapshot** | **4** |
| **kslwtbctx** | **4** |
| **kslwtectx** | **4** |
| opikndf2 | 4 |

# 神奇的等待事件

这其中**kslwtbctx**是最早被调用的函数:

| | |
|---|---|
| KGHISPIR | 4 |
| _save_nv_regs | 4 |
| kews_sqlcol_begin | 4 |
| kews_update_wait_time | 4 |
| kghxhal | 4 |
| kghxhfr | 4 |
| kglGetMutex | 4 |
| kksGetStats | 4 |
| **kskthbwt** | **4** |
| **kskthewt** | **4** |
| **kslwt_end_snapshot** | **4** |
| **kslwt_start_snapshot** | **4** |
| **kslwtbctx** | **4** |
| **kslwtectx** | **4** |
| opikndf2 | 4 |

下面，我们就从kslwtbctx开始。

# 神奇的等待事件

我查看了从kslwtbctx开始Oracle会调用的一些函数，它们依次是：

```
i=520 kslwtbctx(fffffd7fffdfbl80,l,fffffd7fffdfb3d7,l,c725l58,4034);
i=52l gethrtime(2l,l,fffffd7fffdfb3d7,l,ceed648,0);
i=522 kskthbwt(0,42beedl3,742beedl3,4c5e2df93bee,3ff0,395c42ba8);
i=523 memcpy(395be69b0,fffffd7fffdfble8,30,7b,c725l58,395c42ba8);

     0  l  2  3  4  5  6  7  8  9  a  b  c  d  e  f  0l23456789abcdef
 0: 55 0l 00 00 00 00 00 00 ff ff ff 7f 7f fd ff ff  U...............

i=524 kslwt_start_snapshot(395be6948,395be6948,l,7b,ced88lc,ceed648);
i=525 nioqsn(cedda60,0,fffffd7fffdfb3d7,l,380009ce8,ceed648);
```

……

Oracle在第520次函数调用时，调用了kslwtbctx，在第521次调用了gethrtime，这是一个获取时间的函数。等待事件的一个重要操作，不就是记录时间吗！

补充一点，观察内存的流动很重要吗。Memcpy就是完成内存的流动函数。因此观察memcpy拷贝了什么样的值很重要。

# 神奇的等待事件

```
i=520 kslwtbctx(fffffd7fffdfb180,1,fffffd7fffdfb3d7,1,c725158,4034);
i=521 gethrtime(21,1,fffffd7fffdfb3d7,1,ceed648,0);
i=522 kskthbwt(0,42beedl3,742beedl3,4c5e2df93bee,3ff0,395c42ba8);
i=523 memcpy(395be69b0,fffffd7fffdfble8,30,7b,c725158,395c42ba8);

     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  0123456789abcdef
 0: 55 01 00 00 00 00 00 00 ff ff ff 7f 7f fd ff ff  U...............

i=524 kslwt_start_snapshot(395be6948,395be6948,1,7b,ced881c,ceed648);
i=525 nioqsn(cedda60,0,fffffd7fffdfb3d7,1,380009ce8,ceed648);
```

它从fffffd7fffdfb1e8处，向395be69b0拷贝0x30（十进制48）个字节。拷贝的内容我也用DTrace把它显示出来了，“55 01 00 00 00 00 00 00 ff ff ff 7f 7f fd ff ff ……”。

这其中前两个字节“5501”引起了我的注意，我的测试机是小端，5501真正表示的数据是0155，十进制是341.

# 神奇的等待事件

```
i=520 kslwtbctx(fffffd7fffdfbl80,l,fffffd7fffdfb3d7,l,c725l58,4034);
i=52l gethrtime(2l,l,fffffd7fffdfb3d7,l,ceed648,0);
i=522 kskthbwt(0,42beedl3,742beedl3,4c5e2df93bee,3ff0,395c42ba8);
i=523 memcpy(395be69b0,fffffd7fffdfble8,30,7b,c725l58,395c42ba8);

     0  l  2  3  4  5  6  7  8  9  a  b  c  d  e  f  0l23456789abcdef
 0: 55 0l 00 00 00 00 00 00 ff ff ff 7f 7f fd ff ff  U...............

i=524 kslwt_start_snapshot(395be6948,395be6948,l,7b,ced88lc,ceed648);
i=525 nioqsn(cedda60,0,fffffd7fffdfb3d7,l,380009ce8,ceed648);
```

Oracle中的每个等待事件，都有一个编号，这个341会不会就是等待事件编号呢？这很容易验证，查询等待事件编号等于341的，看看到底有没有这个编号，如果有的话，等待事件是什么！

# 神奇的等待事件

查询结果：

```
SQL> select event#, event_id from v$event_name where event#=341;

   EVENT#   NAME
---------- ----------
      341 SQL*Net message to client
```

编号341的等待事件是SQL*Net message to client。

我的测试SQL，会有两次to client等待事件。因此，综合上面这些因素，我更加怀疑kslwtbctx和等待事件一定有关系。哪么，接下来要怎么确认这个猜想呢？

# 神奇的等待事件

还记得前面讲过的“断点”吗？ 我们前面不是还提出过一个问题：如何发现断点吗！Kslwtbctx这些函数，就是我们要找的断点！

找到了断点又有什么用呢？

我们可以让Oracle在断点处停下来，然后我们可以慢慢观察Oracle的状态。

下面，我们就演示一下，我们如何利用断点，挖掘等待事件机制。

# 神奇的等待事件

```
SQL> select event,state from v$session where sid=19;

EVENT                          STATE
------------------------------ -------------------
SQL*Net message from client    WAITED KNOWN TIME
```

kslwtbctx

gethrtime
kskthbwt
memcpy
kslwt_start_snapshot
nioqsn

我先在kslwtbctx函数入口处设置断点，让CPU执行到kslwtbctx函数入口处。我们这里，需要观察的“状态”，就是Oracle的等待事件。

可以看到，在CPU执行到kslwttctx入口处时，进程什么都不在等。State列值为“WAITED KNOWN TIME”，这说明进程是ON CPU的状态，当前没有等待事件。

# 神奇的等待事件

```
SQL> select event,state from v$session where sid=19;

EVENT                          STATE
------------------------------ -------------------
SQL*Net message from client    WAITED KNOWN TIME
```

kslwtbctx

**gethrtime**

kskthbwt

memcpy

kslwt_start_snapshot

nioqsn

继续，在下一函数gethrtime处设置断点，让CPU执行到gethrtime函数入口处，观察等待事件，没有变化，进程这时还是没有任何等待事件。

# 神奇的等待事件

```
SQL> select event,state from v$session where sid=19;

EVENT                          STATE
------------------------------ -------------------
SQL*Net message from client    WAITED KNOWN TIME
```

kslwtbctx
gethrtime
**kskthbwt**
memcpy
kslwt_start_snapshot
nioqsn

重复上面的操作，在CPU前进到kskthbwt函数入口处时，还是没有任何等待事件。

# 神奇的等待事件

```
SQL> select event,state from v$session where sid=19;

EVENT                          STATE
------------------------------ -------------------
SQL*Net message from client    WAITING
```

kslwtbctx
gethrtime
kskthbwt
**memcpy**
kslwt_start_snapshot
nioqsn

在CPU前进到memcpy函数入口处时，State列变为了Waiting。

# 神奇的等待事件

```
SQL> select event,state from v$session where sid=19;

EVENT                          STATE
------------------------------ ------------------
SQL*Net message to client      WAITING
```

kslwtbctx
gethrtime
kskthbwt
memcpy
**kslwt_start_snapshot**
nioqsn

在CPU前进到kslwt_start_snapshot函数入口处时，EVENT列变为了”SQL*Net message to client”。到此，Oracle完成了“登记”等待事件这一操作。
其始从kslwtbctx处，等待事件就已经开始了。它调用gethrtime得到时间，然后调用kskthbwt修改v$session.STATE列状态。再接着调用memcpy函数，将等待事件编号拷贝到SGA中的v$session中。

# 神奇的等待事件

我们已经知道了kslwtbctx是Oracle产生等待事件的函数，哪么，等待事件的结束函数是什么呢？

这个很容易找到，仍然从调用次数为4的函数中找，很快就能找到，很快就能找到，等待结束的函数是：kslwtectx。下面这一串，是等待事件结束时的函数调用堆栈：

Kslwtectx→
  gethrtime
  kslwt_end_snapshot
  kslwt_update_stats_int
    → kews_update_wait_time
  kskthewt

使用前面的方法，让CPU一个函数一个函数的向下执行，当执行到kskthewt函数时，v$session中state列，会由WAITING变为WAITED KNOWN TIME，说明等待事件到此已经完全结束了。

# 神奇的等待事件

现在，让我们总结一下我们的成果：

1、看到kslwtbctx，就是某个等待事件开始了。红色字母“b”，应该是begin。
2、看到Kslwtectx，就是某个等待事件结束了。红色字母“e”，应该是end。
3、在kslwtbctx之下，会有memcpy，它所拷贝内存的前两个字节，是一个整数（短整型, short int），这个数字就是等待事件编号。

有了这三点信息，我们可以去分析各种各样的等待事件的真正意义。Oracle什么时候会开始一个等待事件，在什么样的条件下，会结束一个等待事件，……。这一切不会花你太多时间。而且，很多时候，还会有意外的发现。

举个简单的小例子，比如等待事件log file switch (private strand flush incomplete)，文档中对它的解释是闪烁其词：
User sessions trying to generate redo, wait on this event when LGWR waits for DBWR to complete flushing redo from IMU buffers into the log buffer; when DBWR is complete LGWR can then finish writing the current log, and then switch log files.

又是LGWR又是DBWR，其实根据我的测试，前台进程在进行DML时，会持有IMU Latch（In memory undo latch），当还没来得急释放IMU Latch时，发生了日志切换，LGWR也要去申请IMU Latch，竞争就出现了。但这时并不会显示IMU Latch等待，而是会显示出log file switch (private strand flush incomplete)。

# 神奇的等待事件

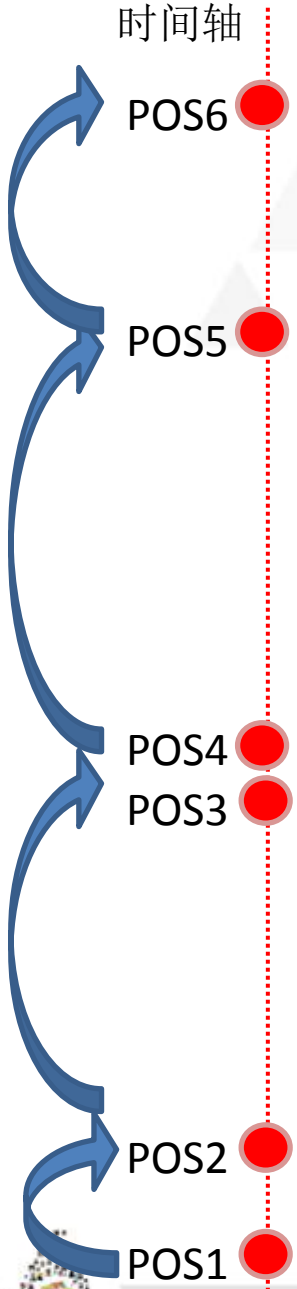
```
i=520 kslwtbctx(fffffd7fffdfbl80,l,fffffd7fffdfb3d7,l,c725l58,4034);
i=52l gethrtime(2l,l,fffffd7fffdfb3d7,l,ceed648,0);
i=522 kskthbwt(0,42beedl3,742beedl3,4c5e2df93bee,3ff0,395c42ba8);
i=523 memcpy(395be69b0,fffffd7fffdfble8,30,7b,c725l58,395c42ba8);

     0  l  2  3  4  5  6  7  8  9  a  b  c  d  e  f  0l23456789abcdef
 0: 55 0l 00 00 00 00 00 00 ff ff ff 7f 7f fd ff ff  U...............

i=524 kslwt_start_snapshot(395be6948,395be6948,l,7b,ced88lc,ceed648);
i=525 nioqsn(cedda60,0,fffffd7fffdfb3d7,l,380009ce8,ceed648);
```

接下来，我重点分析了这个等待事件编号，是何传递到memcpy函数中的。

这里的memcpy函数是由kslwtbctx调用的，下面我们就要结合它的反汇编代码来看了。

但要注意，我们不是要完整阅读kslwtbctx的反汇编代码，因为就算只是Oracle十几万个函数中的一个，它的反汇编代码是相当长的。我们只需要从其中找出来对我们有意义的就可以了。这就简单许多了。

# 神奇的等待事件




```
> kslwtbctx::dis
kslwtbctx:              pushq  %rbp
kslwtbctx+1:            movq   %rsp,%rbp
.....................
kslwtbctx+0x14:         movq   %rdi,%r15
kslwtbctx+0x17:         movq   0xb9003c2(%rip),%r8
kslwtbctx+0x1e:         movq   %r8,-0x138(%rbp)
.....................
kslwtbctx+0x205:        movq   %r13,0x18(%r15)
kslwtbctx+0x209:        movq   $0x0,0x28(%r15)
kslwtbctx+0x211:        movw   $0x0,0x30(%r15)
kslwtbctx+0x218:        leaq   0x68(%r15),%r8
kslwtbctx+0x21c:        movq   %r8,-0x150(%rbp)
kslwtbctx+0x223:        movslq 0x68(%r15),%r8
kslwtbctx+0x227:        imulq  $0x30,%r8,%r8
.....................
kslwtbctx+0x3f2:        movq   %r8,0x60(%r14)
kslwtbctx+0x3f6:        leaq   0x68(%r14),%rdi
kslwtbctx+0x3fa:        movq   -0x150(%rbp),%rsi
kslwtbctx+0x401:        movq   $0x30,%rdx
kslwtbctx+0x408:  call  -0x112e58       <PLT=libc.so.1`memcpy>
```

64位系统中，调用函数时，第一个参数在rdi中，第二个参数在rsi中……

# 神奇的等待事件

我们最终的结论，kslwtbctx函数的第一个参数中，记录有等待事件信息。而且，具体位置是第一个参数向下0x68字节处，记录着等待事件编号。

# 案例：从调用堆栈挖掘等待事件

基础知识也介绍的差不多了，下面我们我们来看一个案例吧。案例很简单，一套RAC库，11.1.0.7，某些节点最近一段时间偶而DOWN机。最近的一次当机，产生如下DTRAE文件：

Dump continued from file: /opt/oracle/diag/rdbms/orcl/orcl3/trace/orcl3_lms0_13659.trc
ORA-07445: exception encountered: core dump [skgxpdmpmem()+52481] [SIGSEGV] [ADDR:0x1FFFFFFFFD5F9478] [PC:0xC00000000DC43651] [Address not mapped to object] []

这几行说明更进一步的信息，在LMS进程的TRC文件。我们可以打开orcl3_lms0_13659.trc，进一步分析问题。

# 案例：从调用堆栈挖掘等待事件

这是orcl3_lms0_13659.trc中的内容，可以看到LMS进程的Session ID是449：

*** SESSION ID:(449.1)

接下来，用会话号449，在文件内搜索，可以找到LMS进程最后的等待事件是什么：

```
----------------------------------------
 SO: 0xc00000123065dcc8, type: 4, owner: 0xc00000123000a0c8, flag: INIT/-/-/0x00 if: 0x3
c: 0x3
     proc=0xc00000123000a0c8, name=session, file=ksu.h LINE:10719 ID:, pg=0
    (session) sid: 449 ser: 1 trans: 0x0000000000000000, creator: 0xc00000123000a0c8
ksuxds FALSE at location: 0
    service name: SYS$BACKGROUND
    Current Wait Stack:
     0: waiting for 'gcs remote message'
Wait State:
      auto_close=0 flags=0x22 boundary=0x0000000000000000/-1
    Session Wait History:
     0: waited for 'gcs remote message'
     1: waited for 'gcs remote message'
     2: waited for 'gcs remote message'
     3: waited for 'gcs remote message‘
.....
```

# 案例：从调用堆栈挖掘等待事件

这是orcl3_lms0_13659.trc中的内容，可以看到LMS进程的Session ID是449：

*** SESSION ID:(449.1)

接下来，用会话号449，在文件内搜索，可以找到LMS进程最后的等待事件是什么：

---

SO: 0xc00000123065dcc8, type: 4, owner: 0x... 0x3
c: 0x3
proc=0xc00000123000a0c8, name=session, file... LINE:10719 ID:, pg=0
(session) sid: 449 ser: 1 trans: 0x0000000000...00, creator: 0xc00000123000a0c8
ksuxds FALSE at location: 0
service name: SYS$BACKGROUND
Current Wait Stack:
0: waiting for 'gcs remote message'
Wait State:
auto_close=0 flags=0x22 boundary=0x0000000000000000/-1
Session Wait History:
0: waited for 'gcs remote message'
1: waited for 'gcs remote message'
2: waited for 'gcs remote message'
3: waited for 'gcs remote message'
.....

它最后在等'gcs remote message

# 案例：从调用堆栈挖掘等待事件

但是，接下来查看调用堆栈的时候，发现了另外的信息：

```
----- Call Stack Trace -----
calling                  call     entry                argument values in hex
location                 type     point                (? means dubious value)
------------------------ -------- -------------------- ----------------------------
.....................
kslwaitctx()+240         call     $cold_ksliwat()      C00000123065F668 ?
                                                       C00000123065F668 ?
                                                       000000003 ?
                                                       600000000013F700 ?
kslwait()+192            call     kslwaitctx()         9FFFFFFFFFFFB710 ?
                                                       000000003 ?
.....................
.....................
main()+224               call     opimai_real()        000000003 ? 000000000 ?
main_opd_entry()+80      call     main()               000000003 ?
                                                       9FFFFFFFFFFFF820 ?
                                                       600000000013F700 ?
                                                       C000000000000004 ?
```

以前调用堆栈这块，我是从来不看的。开搞“调试Oracle”以后，经常能从里面发现一些老熟人，这才发现，其实里面包含了重要信息。

# 案例：从调用堆栈挖掘等待事件

这次碰到的老熟人是它：

```
----- Call Stack Trace -----
calling                  call     entry                argument values in hex
location                 type     point                (? means dubious value)
------------------------ -------- -------------------- ----------------------------
.....................
kslwaitctx()+240         call     $cold_ksliwat()      C00000123065F668 ?
                                                       C00000123065F668 ?
                                                       000000003 ?
                                                       600000000013F700 ?
kslwait()+192            call     kslwaitctx()         9FFFFFFFFFFFB710 ?
                                                       000000003 ?
.....................
.....................
main()+224               call     opimai_real()        000000003 ? 000000000 ?
main_opd_entry()+80      call     main()               000000003 ?
                                                       9FFFFFFFFFFFF820 ?
                                                       600000000013F700 ?
                                                       C000000000000004 ?
```

kslwaitctx”函数是11GR1版本中，登记等待事件的函数。它就是11GR2 中的kslwtbctx。关于这个函数，我之前恰好研究过它。

# 案例：从调用堆栈挖掘等待事件

再来看TRC文件中的调用堆栈：

```
----- Call Stack Trace -----
calling                  call     entry                argument values in hex
location                 type     point                (? means dubious value)
------------------------ -------- -------------------- ----------------------------
.....................
kslwaitctx()+240         call     $cold_ksliwat()      C000001230 65F668 ?
                                                       C000001230 65F668 ?
                                                       000000003 ?
                                                       600000000013F700 ?
kslwait()+192            call     kslwaitctx()         9FFFFFFFFFFFB710 ?
                                                       000000003 ?
.....................
.....................
main()+224               call     opimai_real()        000000003 ? 000000000 ?
main_opd_entry()+80      call     main()               000000003 ?
                                                       9FFFFFFFFFFFF820 ?
```

“9FFFFFFFFFFFB710”，就是kslwaitctx的第一个参数。 kslwaitctx 就是11GR1中的kslwaitctx。还记得我们刚才分析的，它的第一个参数有什么意义吗？

“第一个参数向下0x68字节处，记录着等待事件编号”，这是我们刚才的分析结论。

# 案例：从调用堆栈挖掘等待事件

“9FFFFFFFFFFFB710”，就是kslwaitctx的第一个参数：

```
----- Call Stack Trace -----
calling               call     entry                argument values in hex
location              type     point                (? means dubious value)
--------------------- -------- -------------------- ----------------------------
.................
kslwaitctx()+240      call     $cold_ksliwat()      C00000123065F668 ?
                                                    C00000123065F668 ?
                                                    000000003 ?
                                                    600000000013F700 ?
kslwait()+192         call     kslwaitctx()         9FFFFFFFFFFFB710 ?
                                                    000000003 ?
.................
.................
main()+224            call     opimai_real()        000000003 ? 000000000 ?
main_opd_entry()+80   call     main()               000000003 ?
                                                    9FFFFFFFFFFFF820 ?
```

也就是说， 0x9FFFFFFFFFFFB710+0x68处，就是等待事件的编号。我们是否能得到此处的值是什么呢？没问题，Oracle不仅会把调用堆栈放到TRC文件中，而且每个函数参数所指向的内存，也都会被DUMP到TRC文件中。以“9FFFFFFFFFFFB710”为关键字在TRC文件中搜索，就能得到结果。

# 案例：从调用堆栈挖掘等待事件

下面蓝底的部分，就是“9FFFFFFFFFFB710”处的内存值了。 9FFFFFFFFFFB710向下0x68（十进制104）字节处的值： 000000A0。它就是等待事件的event#，十进制是160。在相同的版本下（这个库版本是11.1.0.7），在v$event_name查看event#为160的等待事件：gc current block lost。

```
713 Argument/Register addr=0x9ffffffffffb710.
714 Dump of memory from 0x9ffffffffffb6d0 to 0x9ffffffffffb810
715 9FFFFFFFFFFB6D0 00000000 00136B44 20B747A3 0A7F143C  [......kD .G....<]
716 9FFFFFFFFFFB6E0 00000000 0001003E 346DC5D6 3886594B  [.......>4m..8.YK]
717 9FFFFFFFFFFB6F0 9FFFFFFF FFFFB7DC 00000010 00000000  [................]
718 9FFFFFFFFFFB700 00000000 0001003E 346DC5D6 3886594B  [.......>4m..8.YK]
719 9FFFFFFFFFFB710 7A590000 307446D2 00000089 870042D1  [zY..0tF.......B.]
720 9FFFFFFFFFFB720 00000089 870042D1 00000089 870042D1  [......B.......B.]
721 9FFFFFFFFFFB730 00000089 870042D1 00000000 00000000  [......B.........]
722 9FFFFFFFFFFB740 00000000 00000000 00000089 87004290  [..............B.]
723 9FFFFFFFFFFB750 00000089 870042A8 00000000 00000000  [......B.........]
724 9FFFFFFFFFFB760 00000003 000005A8 00000001 00000000  [................]
725 9FFFFFFFFFFB770 40000000 019172B0 000000A0 00000000  [@.....r.........]
726 9FFFFFFFFFFB780 00000003 00000006 00000000 00000018  [................]
727 9FFFFFFFFFFB790 00000000 00000000 00000000 00000000  [................]
728         Repeat 1 times
```

此处就是等待事件编号

# 案例：从调用堆栈挖掘等待事件

为什么TRC文件下面的等待事件是gcs remote message，而我们从调用堆栈挖出的等待事件是gc current block lost？

具体原因，就说来话长了，这里就不展开描述了。不过，这样的情况是很少见的，大部分时候调用堆栈中的等待事件，和下面DUMP的等待事件是一样的。

但如果出现不一样的情况，我认为应该以调用堆栈中的等待事件，做为最后的等待事件

哪么，这里的问题最后是如何解决的呢？

# 案例：从调用堆栈挖掘等待事件

gc current block lost，这是一个相关网络的等待事件，之所以会LOST，通常都是因为网络的问题。

但是这里的网络，并无明显问题，相关人员也没有进一步排查的欲望。问题还只能我们DBA从数据库端解决。

普通DML所引发的gc current block lost类等待，就算LOST问题也不会太大，顶多就是某一个进程报错。这里，既然已经引起宕机了，问题肯定不是DML，最可能是DDL引起的，而且应该是频繁执行的DDL。按照这个方向，对数据库SQL进行排查，发现有一个truncate，被频繁的执行。和开发协商后，改为临时表后，节点不再DOWN掉。

# 广告（一）

简单介绍一下我们公司的数据库团队，我们未来可能会开放一个DBA职位。

我们是一个国际化的团队，一半以上的同事在美国。我们管理着ebay所有的OLTP数据库，包括MySQL、Mongo、 cassandra、couchbase，当然还有Oracle。

我们最大的一套OLTP Oracle数据库（之所以用“套”，因为我们也进行了拆分），有超过600TB数据量，正常负载下，每秒事务数5到8万次，是我遇到过的最繁忙的一套OLTP系统。

成长与挑战并存，如果你希望应对挑战，到时候可以联系我：

ITPUB ID：VAGE

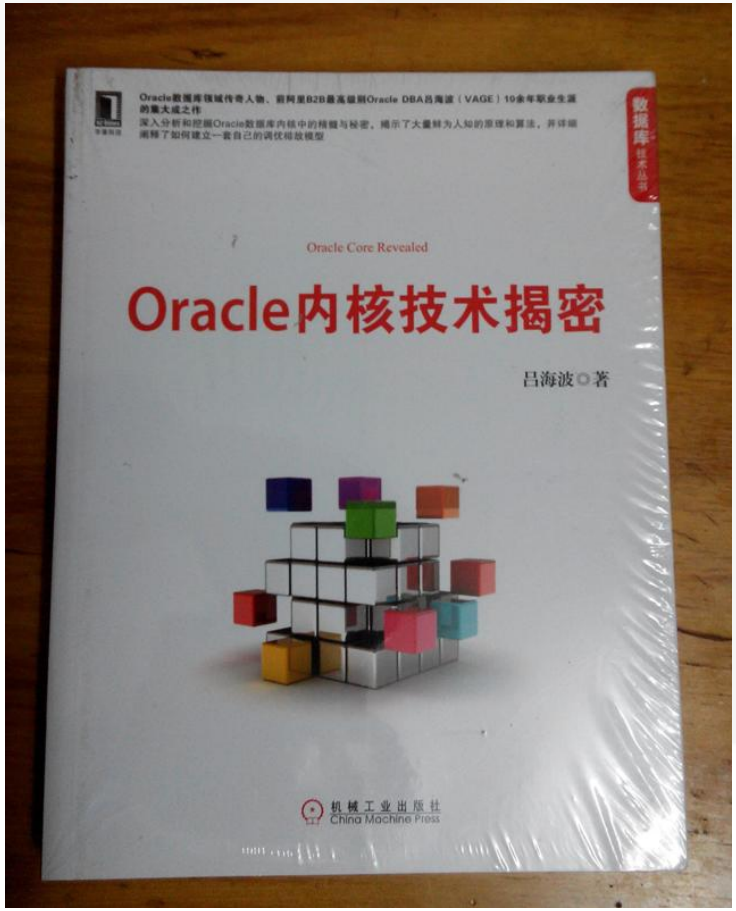
微博：文本时代_VAGE

vage

中国 上海

扫一扫上面的二维码图案，加我微信

# 广告（二）




居家旅行杀人灭口必备良药。因为时间苍促，错误之处在所难免，拍砖的时候手下留情。


ChinaUnix
ITPUB
THANKS