打 造 操 作 系 统 创 新 生 态

庖丁解牛十三刀

作　者：周 鹏

日　期：2021年6月

引言

无论是性能优化还是疑难问题解决，学会用工具再加上自己的独特方法常常助我们快速找到问题的根因、快速找到调用的关键函数、快速分析系统的机制等等。

这些方法可以很多，本文详细介绍了除内存泄露之外的常用调试工具和调试技巧，为了方便记忆，总结成了十二种方法，重点介绍如何在实际的工作中将工具和独特的方法结合起来。

所以本文不是简单的介绍这些工具、方法。本文的思路是先介绍工具的原理、什么时候用它、然后通过具体的实例来分析是如何巧妙的用这些工具快速分析问题。

这些方法也不是孤立使用，当前也不是全部一起使用，通常是几种方法混合使用，如gdb+bfp+strace+文件系统节点+反汇编+perf。到底用哪几种，还需要我们自己撑握、实战后灵活使用。

统信软件 UNIONTECH

01. GDB
02. BPF
03. ftrace
04. strace
05. vmtouchX
06. perf
07. 文件系统节点调试
08. 定制控制代码调试
09. 互换内核调试
10. 汇编调试
11. 预编译调试
12. 页保护原理调试
13. valgrind

统信软件 UNIONTECH

1. GDB

统信软件 UNIONTECH

1 引言

GDB最强大的功能在用它的调试跟踪能力

对于用户态调试，我们常用的功能有(但远不止这些)：
查看堆栈
单步跟踪
反汇编调试

查看堆栈可以用来分析流程和死机问题，分析流程时通过堆栈可以知道调用关系；分析死机问题时，跟踪堆栈可以知道最后的死机函数，分析此函数再结合单步跟踪通常来快速找到问题根因。

单步跟踪非常灵活，可以查看变量的值，可以通过修改变量的值改变代码的运行流程，可以在单步运行中了解一下数据结构的内容是如何发生变化的。

反汇编调试通常用在非调试版本或者是帮助理解内核代码

对于内核态调试，我们常用的功能有：
查看全局变量的值，包括可获取得的内存地址的值
反汇编内核代码

统信软件
UNIONTECH

1.1 gdb常用命令

run:　　　　运行程序

gdb process -p pid: 调试一个已经启动的进程

breakboint :设置断点

　　b funcname :针对函数名设置断点

　　b file:line　:针对文件名和行号设置断点，通常用于调试版本

　　b *addr　:针对地址设置断点，通常用于汇编调试时设置断点

　　b file:line if var==12 :设置条件断点,当var==12时成会停下来

　　b file:line if t=5:只有线程5运行到这儿段点时停下来

print:　　打印变量 的值

　　p /x /d varr　:打印变量的值，/x表示以16进制格式查看，/d表示十进制格式查看, 不要同时使用

　　p /x /d array[0]@len :打印数组

　　p (char*)addr: 将某个地址类容强制转换成字符串输出

　　p ((long*)addr)[0]@len :将某个地址强制转换成long方式查看，如addr中保存的是一组64位地址。

　　p varr=value :改写某变量的值

　　p array[i]=value :也可以改写数组某项的值

　　p $x0　　:打印x0宏存器的值

统信软件 UNIONTECH

1.1.2 gdb常用命令

call:调用某个函数

p errno

call strerror(19) :假设获取到的errno=19, call strerror可以获取到系统给的错误信息

call malloc(24)   :动态申请一段内存,如发现出现了内存改写，看看是否能再次触发异常

call dumpDoxTree(&node) :动态传入一个dom树的地址打印dom树，假设写了这个打印函数

next:执行一行代码，如果是一个函数调用，执行完该函数，如果非调试版本无法有效使用

step into:执行一行代码，如果是一个函数调用，进入该函数,如果非调试版本无法有效使用

stepi: 执行一条汇编指令

display /x /d var:

jump:跳转到某行或某个地址，适用要重新执行某个代码块

until: 一直执行到某行或某个地址

continue:适用于继续执行，一直到遇到下一个断点

disasamble:反汇编

disas func:反汇编某个函数

disas addr1, addr2 :反汇编某个代码段

统信软件 UNIONTECH

1.1.3 gdb常用命令

display:显示某个变量的值,每执行一条命令就显示一下

display /x *(int*)0xfffff7600000

调试一行就显示一个这个内存的值,这个在观察此变量的值什么时候发生变化很有用

examine memory:检查某个内存的值. 查一段的值,检查某个地址是否有效时很有用

x /32w 0xfffff7600000 检查这个地址的值,如果是变量的地址,则以4B为单位,显示32个连续内存的值

x myaddr 检查这个变量的值

x 0xfffff7ec1f78 假设这个地址是某个函数的地址,那么通常会这么显示

0xfffff7ec1f78 <__libc_open64>: 0xa9b97bfd

info reg: 显示所有宏存器的值

backtrace: 查看堆栈

thread apply all bt: 查看整个进程的堆栈

thread n: 显示切换到某个线程,如切换过去后再执行bt查看堆栈

up/down: 显示堆栈后调用up切换到上一层函数,down则回到下一层函数. 切换到哪一层就能看哪一层的局部变量的值.

1.2 查看堆栈的简单代码

```
#include <string.h>
void test3()
{
    char acBuf[12];
    memcpy(acBuf, "123456", 10);
    strcpy(0, "abcdefg");
}

void test2()
{
    test3();
}

void test1()
{
    test2();
}

int main(int argc, char** argv)
{
  test1();
    return 0;
}
```

代码很简单，写test3, test2, test1是为了展示堆栈

统信软件 UNIONTECH

1.2.1 代码解释

```
#include <string.h>
void test3()
{
    char acBuf[12];
    memcpy(acBuf, "123456", 10);
    strcpy(0, "abcdefg");
}

void test2()
{
    test3();
}
```

1)人为构造出一个地址访问异常的段错误，看看gdb是如何反馈这个段错误的

```
void test1()
{
    test2();
}

int main(int argc, char** argv)
{
  test1();
    return 0;
}
```

统信软件 UNIONTECH

1.2.2 查看堆栈

arm平台

```
Program received signal SIGSEGV, Segmentation fault.
__memcpy_generic () at ../sysdeps/aarch64/multiarch/../memcpy.S:127
127     ../sysdeps/aarch64/multiarch/../memcpy.S: 没有那个文件或目录.
(gdb) bt
#0  __memcpy_generic () at ../sysdeps/aarch64/multiarch/../memcpy.S:127
#1  0x00000000004005b0 in test3 () at gdbstack.c:3
#2  0x00000000004005c8 in test2 () at gdbstack.c:8
#3  0x00000000004005e0 in test1 () at gdbstack.c:13
#4  0x0000000000400600 in main (argc=1, argv=0xfffffffdf58) at gdbstack.c:18
(gdb)
```

1)bt命令查看堆栈

2)死机的地址是memcpy函数，c库函数本身是不会有问题的，肯定是传入参数问题

up 堆栈查看死机函数，可以看到导致死机的代码就是strcpy的调用，死机的原因是将”123456”字符串写到一个无效地址0出现了段错误

```
(gdb) up
#1  0x00000000004005b0 in test3 () at gdbstack.c:3
3               strcpy(0, "123456");
(gdb)
```

3)使用up函数往上跟踪，查看调用memcpy的地方。

4) 是strcpy调用引起，c库在底层用memcpy实现strcpy

统信软件
UNIONTECH

1.3 修改变量控制代码运行逻辑

将test函数如下修改，which变量为不同的值时就可以控制代码的逻辑。

```
1  void test3(int which)
2  {
3       char acBuf[16];
4       if (0 == which) {
5           strcpy(0, "123456");
6       }
7       else {
8           strcpy(acBuf, "123456");
9       }
10 }
```

1)调用test3时，which=0

2)默认走这个分支

3） 当将set which=1时，强制走到这个分支

gdb中如下操作:

```
Breakpoint 1, test3 (which=0) at gdbstack.c:4
4               if (0 == which) {
(gdb) set which=1
(gdb) p which
$1 = 1
(gdb) n
8                   strcpy(acBuf, "123456");
(gdb)
```

1)强制修改为1

2） 查看修改后的值

3） 修改成功，走else分支了

统信软件 UNIONTECH

1.4 汇编调试

test3函数如下修改：

memcpy(acBuf, "123456", 10);

strcpy(0, "abcdefg");

A)memcpy(acBuf, "123456", 10); 反汇编进入。首次调用memcpy函数时，需要通过memcpy@plt函数计算出memcpy函数的真正地址。

```
Dump of assembler code for function test3:
   0x0000000000400594 <+0>:     stp     x29, x30, [sp, #-32]!
   0x0000000000400598 <+4>:     mov     x29, sp
   0x000000000040059c <+8>:     add     x3, sp, #0x10
=> 0x00000000004005a0 <+12>:    mov     x2, #0xa                 // #10
   0x00000000004005a4 <+16>:    adrp    x0, 0x400000
   0x00000000004005a8 <+20>:    add     x1, x0, #0x6d8
   0x00000000004005ac <+24>:    mov     x0, x3
   0x00000000004005b0 <+28>:    bl      0x400450 <memcpy@plt>
   0x00000000004005b4 <+32>:    mov     x2, #0x8                 // #8
   0x00000000004005b8 <+36>:    adrp    x0, 0x400000
   0x00000000004005bc <+40>:    add     x1, x0, #0x6e0
   0x00000000004005c0 <+44>:    mov     x0, #0x0                 // #0
   0x00000000004005c4 <+48>:    bl      0x400450 <memcpy@plt>
   0x00000000004005c8 <+52>:    nop
   0x00000000004005cc <+56>:    ldp     x29, x30, [sp], #32
   0x00000000004005d0 <+60>:    ret
End of assembler dump.
```

1)设置参数，memcpy函数需要参数

2) 无法直接调用memcpy, 因为因为不知道此函数的地址，只能通过重定位函数memcpy@plt来找memcpy

```
0000000000400430 <.plt>:
  400430:       a9bf7bf0        stp     x16, x30, [sp, #-16]!
  400434:       90000090        adrp    x16, 410000 <__FRAME_END__+0xf778>
  400438:       f947fe11        ldr     x17, [x16, #4088]
  40043c:       913fe210        add     x16, x16, #0xff8
  400440:       d61f0220        br      x17
  400444:       d503201f        nop
  400448:       d503201f        nop
  40044c:       d503201f        nop

0000000000400450 <memcpy@plt>:
  400450:       b0000090        adrp    x16, 411000 <memcpy@GLIBC_2.17>
  400454:       f9400211        ldr     x17, [x16]
  400458:       91000210        add     x16, x16, #0x0
  40045c:       d61f0220        br      x17
```

2).plt的函数地址是0x400430

1)memcpy@plt的函数地址是0x400450

统信软件 UNIONTECH

1.4.2 汇编调试

B)理解memcpy@plt函数的实现

memcpy@plt函数进一步调用.plt函数。 .plt函数是一个公共函数，每个xxx@plt函数都会去找.plt函数进行重定位，对于memcpy，重定位后的函数地址就要保存到0x411000这个地址空间了

```
   0x00000000004005c4 <+48>:    bl      0x400450 <memcpy@plt>
   0x00000000004005c8 <+52>:    nop
   0x00000000004005cc <+56>:    ldp     x29, x30, [sp], #32
   0x00000000004005d0 <+60>:    ret
End of assembler dump.
(gdb) si
0x00000000004005a4      6           memcpy(acBuf, "123456", 10);
(gdb)
0x00000000004005a8      6           memcpy(acBuf, "123456", 10);
(gdb)
0x00000000004005ac      6           memcpy(acBuf, "123456", 10);
(gdb) si
0x00000000004005b0      6           memcpy(acBuf, "123456", 10);
(gdb) si
0x0000000000400450 in memcpy@plt ()
(gdb) disas
Dump of assembler code for function memcpy@plt:
=> 0x0000000000400450 <+0>:     adrp    x16, 0x411000 <memcpy@got.plt>
   0x0000000000400454 <+4>:     ldr     x17, [x16]
   0x0000000000400458 <+8>:     add     x16, x16, #0x0
   0x000000000040045c <+12>:    br      x17
End of assembler dump.
(gdb) si
0x0000000000400454 in memcpy@plt ()
(gdb) x /x 0x411000
0x411000 <memcpy@got.plt>:      0x00400430
(gdb) si
0x0000000000400458 in memcpy@plt ()
(gdb) si
0x000000000040045c in memcpy@plt ()
(gdb) p /x $x17
$1 = 0x400430
(gdb)
```

1)x16=0x411000, 从这个地址得到.plt函数的地址

5) .plt函数重定位后写此地址

2).plt函数的地址是0x400430

3)x17中保存的已经是.plt函数的地址了

```
gdbstack
uos@uos-PC:~/zhoupeng/train/method1$ cat /proc/25315/maps
00400000-00410000 r-xp 00000000 08:24 808001543                          /home/uos/zhoupeng/train/method1/gdbstack
00410000-00420000 rw-p 00000000 08:24 808001543                          /home/uos/zhoupeng/train/method1/gdbstack
fffff7e00000-fffff7f60000 r-xp 00000000 08:23 503317549                  /usr/lib/aarch64-linux-gnu/libc-2.28.so
fffff7f60000-fffff7f70000 rw-p 00150000 08:23 503317549                  /usr/lib/aarch64-linux-gnu/libc-2.28.so
fffff7f70000-fffff7f80000 rw-p 00000000 00:00 0
fffff7fa0000-fffff7fb0000 r--p 00000000 00:00 0                          [vvar]
fffff7fb0000-fffff7fc0000 r-xp 00000000 00:00 0                          [vdso]
fffff7fc0000-fffff7fe0000 r-xp 00000000 08:23 503317544                  /usr/lib/aarch64-linux-gnu/ld-2.28.so
```

4)0x411000区是进程的可写地址区，可写是为了后面的重定位

```
0000000000400430 <.plt>:
  400430:       a9bf7bf0        stp     x16, x30, [sp, #-16]!
  400434:       90000090        adrp    x16, 410000 <__FRAME_END__+0xf778>
  400438:       f947fe11        ldr     x17, [x16, #4088]
  40043c:       913fe210        add     x16, x16, #0xff8
  400440:       d61f0220        br      x17
  400444:       d503201f        nop
  400448:       d503201f        nop
  40044c:       d503201f        nop

0000000000400450 <memcpy@plt>:
  400450:       b0000090        adrp    x16, 411000 <memcpy@GLIBC_2.17>
  400454:       f9400211        ldr     x17, [x16]
  400458:       91000210        add     x16, x16, #0x0
  40045c:       d61f0220        br      x17

0000000000400460 <__libc_start_main@plt>:
  400460:       b0000090        adrp    x16, 411000 <memcpy@GLIBC_2.17>
  400464:       f9400611        ldr     x17, [x16, #8]
  400468:       91002210        add     x16, x16, #0x8
```

5).plt函数的地址=0x400430

1.4.3 汇编调试

c).plt函数首次执行_dl_runtime_resolve函数

```
0x00000000004005ac      6           memcpy(acBuf, "123456", 10);
(gdb)
0x00000000004005b0      6           memcpy(acBuf, "123456", 10);
(gdb)
0x0000000000400450 in memcpy@plt ()
(gdb)
0x0000000000400454 in memcpy@plt ()
(gdb)
0x0000000000400458 in memcpy@plt ()
(gdb)
0x000000000040045c in memcpy@plt ()
(gdb)
0x0000000000400430 in ?? ()
(gdb) si
0x0000000000400434 in ?? ()
(gdb)
0x0000000000400438 in ?? ()
(gdb)
0x000000000040043c in ?? ()
(gdb) p /x $x17
$16 = 0xffff7fd2b4c
(gdb) x /x $x17
0xfffff7fd2b4c <_dl_runtime_resolve>:    0xa9b327e8
(gdb) p /x $x16
$17 = 0x410000
(gdb) p /x ($x16+4088)
$18 = 0x410ff8
(gdb)
```

```
67   [21] .got              PROGBITS         0000000000410fd8  00000fd8
68        0000000000000010  0000000000000008  WA       0     0     8
69   [22] .got.plt          PROGBITS         0000000000410fe8  00000fe8
70        0000000000000038  0000000000000008  WA       0     0     8
71   [23] .data             PROGBITS         0000000000411020  00001020
72        0000000000000010  0000000000000000  WA       0     0     8
73   [24] .bss              NOBITS           0000000000411030  00001030
74        0000000000000008  0000000000000000  WA       0     0     1

Disassembly of section .plt:

0000000000400430 <.plt>:
  400430:       a9bf7bf0        stp     x16, x30, [sp, #-16]!
  400434:       90000090        adrp    x16, 410000 <__FRAME_END__+0xf778>
  400438:       f947fe11        ldr     x17, [x16, #4088]
  40043c:       913fe210        add     x16, x16, #0xff8
  400440:       d61f0220        br      x17
  400444:       d503201f        nop
  400448:       d503201f        nop
  40044c:       d503201f        nop

0000000000400450 <memcpy@plt>:
  400450:       b0000090        adrp    x16, 411000 <memcpy@GLIBC_2.17>
  400454:       f9400211        ldr     x17, [x16]
  400458:       91000210        add     x16, x16, #0x0
  40045c:       d61f0220        br      x17

0000000000400460 <__libc_start_main@plt>:
```

1)x16=0x410000，到0x410ff8中取需要执行的函数，它就是_dl_runtime_resolve

2)去执行_dl_runtime_resolve函数

统信软件 UNIONTECH

1.4.4 汇编调试

D)最终结果

.plt函数运行_dl_runtime_resolve函数后，将_memcpy_generic这个函数的地址更新到了0x411000这个内存空间

```
1: /x ((unsigned long*)(0x411000))[0] = 0x400430
(gdb)
140     in dl-runtime.c
1: /x ((unsigned long*)(0x411000))[0] = 0x400430
(gdb)
142     in dl-runtime.c
1: /x ((unsigned long*)(0x411000))[0] = 0x400430
(gdb)
145     in dl-runtime.c
1: /x ((unsigned long*)(0x411000))[0] = 0x400430
(gdb)
148     in dl-runtime.c
1: /x ((unsigned long*)(0x411000))[0] = 0x400430
(gdb)
_dl_runtime_resolve () at ../sysdeps/aarch64/dl-trampoline.S:101
101     ../sysdeps/aarch64/dl-trampoline.S: 没有那个文件或目录.
1: /x ((unsigned long*)(0x411000))[0] = 0xfffff7e83450
(gdb) x 0xfffff7e83450
0xfffff7e83450 <__memcpy_generic>:      0xf9800020
(gdb)
```

1)dl-runtime.c退出时修改了重定位地址，即改写了0x411000的内容，之前它指向.plt函数，现在它指向_memcpy_generic函数

```
161  0x0000000000000000 (NULL)              0x0
162
163 重定位节 '.rela.dyn' at offset 0x3a0 contains 1 entry:
164   偏移量          信息           类型           符号值          符号名称 + 加数
165 000000410fe0  000300000401 R_AARCH64_GLOB_DA 0000000000000000 __gmon_start__ + 0
166
167 重定位节 '.rela.plt' at offset 0x3b8 contains 4 entries:
168   偏移量          信息           类型           符号值          符号名称 + 加数
169 000000411000  000100000402 R_AARCH64_JUMP_SL 0000000000000000 memcpy@GLIBC_2.17 + 0
170 000000411008  000200000402 R_AARCH64_JUMP_SL 0000000000000000 __libc_start_main@GLIBC
    _2.17 + 0
171 000000411010  000300000402 R_AARCH64_JUMP_SL 0000000000000000 __gmon_start__ + 0
172 000000411018  000400000402 R_AARCH64_JUMP_SL 0000000000000000 abort@GLIBC_2.17 + 0
173
174 The decoding of unwind sections for machine type AArch64 is not currently supported.
175
176 Symbol table '.dynsym' contains 5 entries:
177    Num:    Value          Size Type    Bind   Vis      Ndx Name
178      0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
```

2)readelf看到的信息，它就是一个需要保存重定位函数的地址空间，而且是保存memcpy函数

1.4.5 汇编调试

E)总结

i)首次运行时需要重定位，此时执行过程为：

memcpy --> memcpy@plt --> .plt -> _dl_runtime_resolve --> memcpy_generic

非首次运行时：

memcpy--> memcpy@plt --> memcpy_generic

ii).plt一个公共函数，所有的xxx@plt函数都找此函数完成重定位.

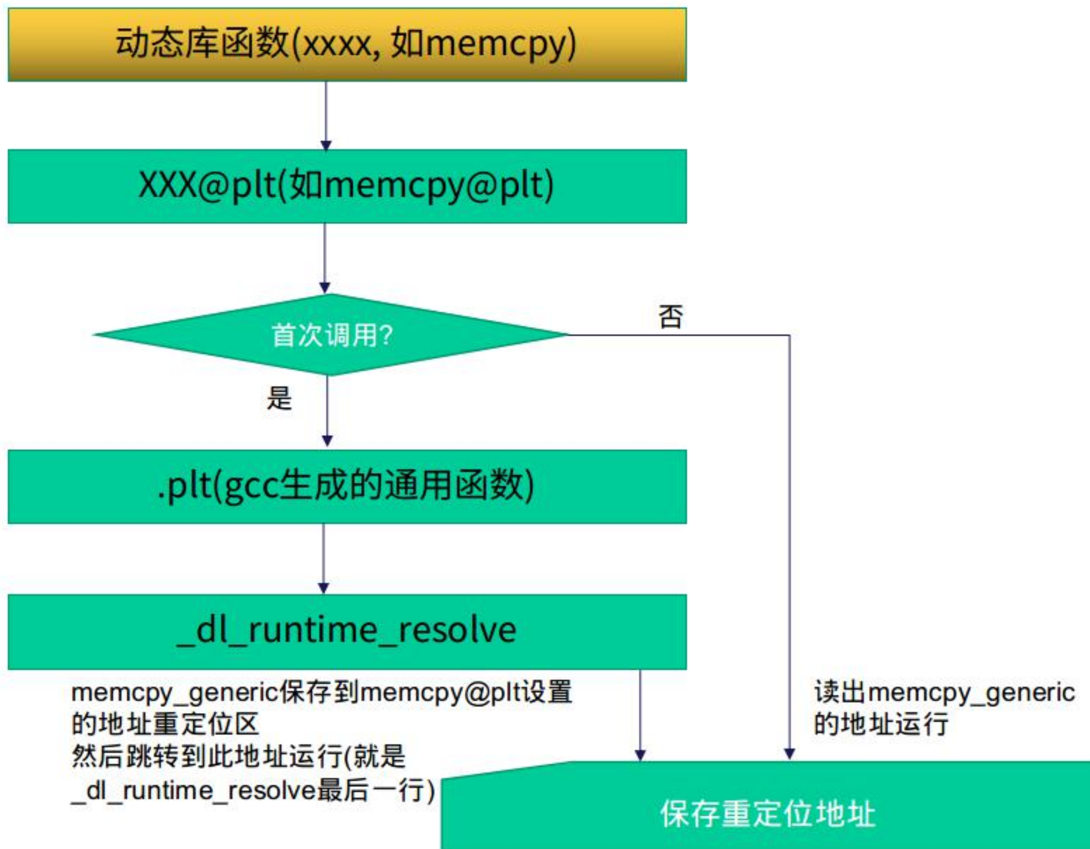
xxx@plt函数找.plt函数重定位时基于x16寄存器给出保存重定位后的地址空间

iii)真正的重定位函数为_dl_runtime_resolve函数，它完成重定位后改写重定位地址空间，这样非首次运行时

memcpy@plt函数就直接调用memcpy_generic了

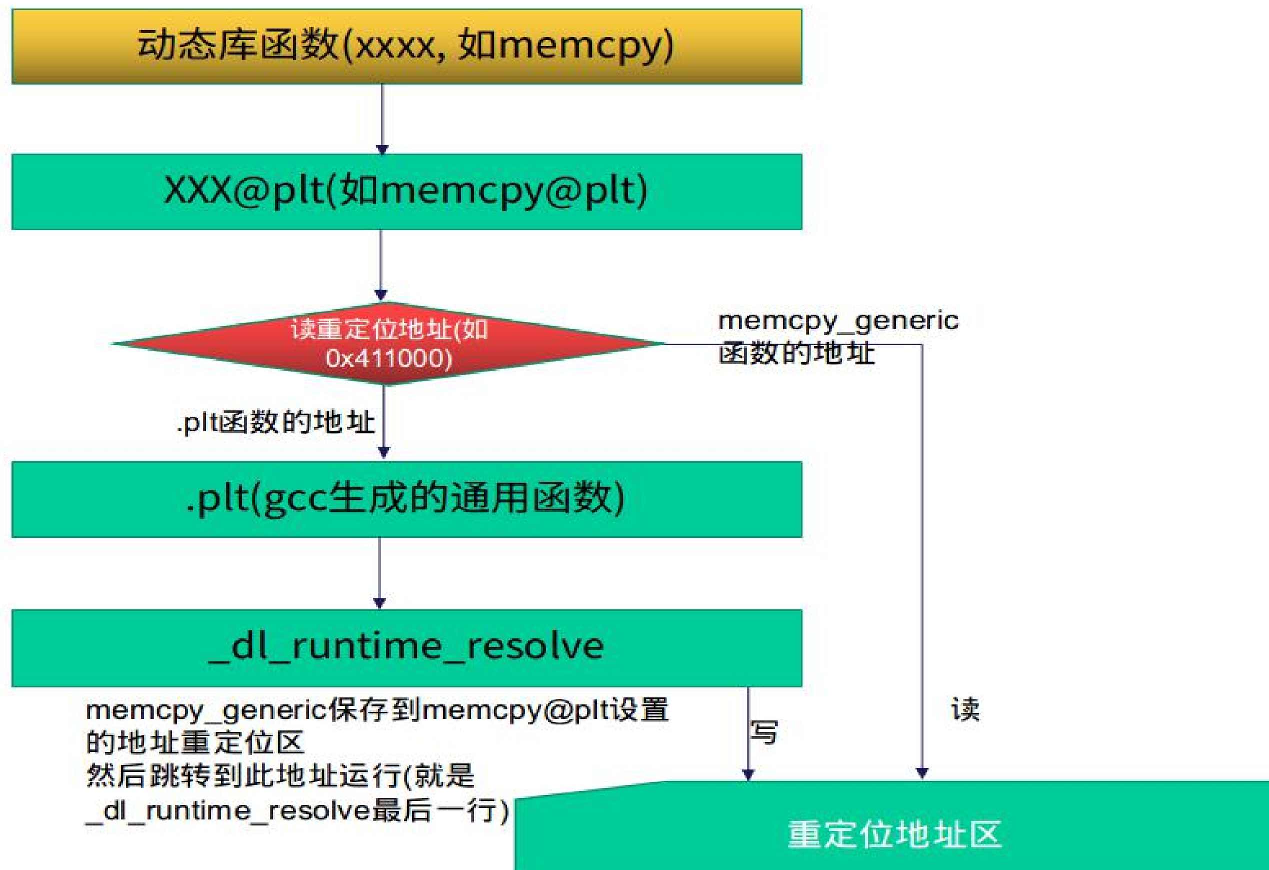
统信软件
UNIONTECH

1.4.6 重定位总结



统信软件
UNIONTECH

1.4.7 重定位总结



统信软件 UNIONTECH

1.5 总结

A）所谓工欲善其事、必先利其器，GDB就是调试的第一大利器，本文没有去全面介绍如何使用GDB的各种命令来调试各种类型的问题，相信有了对工作原理的理解，加上自己的逻辑推理能力，每人都能灵活应用去解决自己的问题，做到无师自通。

B）对于灵活使用，这儿通过调试汇编代码理解了动态库符号的重定位实现，直指本质。本人平时工作中也是根据自己的需求想想如何利用GDB来帮助我们分析问题，很幸运GDB通常都有我需要的功能。

C）从我个人的理解角度来看，GDB就是一个能让程序停下来能让我们随时观察程序内存值的一系列命令的组合。基于停止和观察，可以理解流程、也可以控制流程，从而达到理解代码、理解机制、解决问题。

D）linx上GDB可以协助我们调试C、C++、asm三大语言实现的代码，所以除了可以调试自己写的代码、也可以调试开源代码、内核代码。当面对不是自己写的代码，大量的开源代码特别是C++开源代码混在一起时，GDB的灵活调试优为重要。

E）除了本节讲解的GDB的调试，后面有的章节也会借助GDB来协助调试，总之它一直是我们的得力助手，让我披荆斩棘。

统信软件
UNIONTECH

2. BPF

统信软件
UNIONTECH

2.1 引言

BPF是半个GDB

BPF主要用来调试内核, 由于gdb调试内核的环境搭建很麻烦,而且容易出现死机器, 因此将BPF和gdb结合起来调试内核会非常方便.

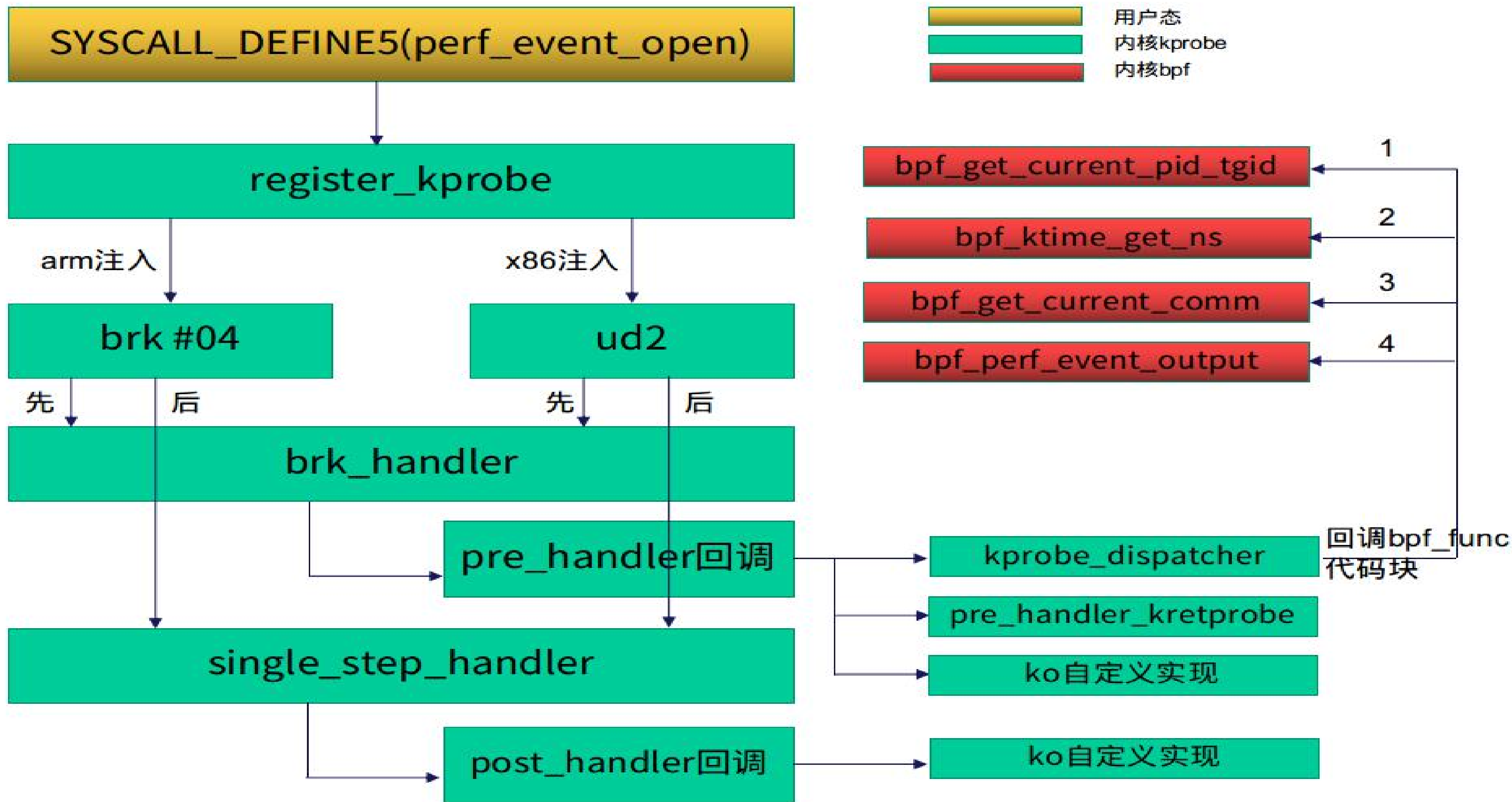
BPF可以用来跟踪函数调用堆栈, 函数调用参数值及函数返回值。

GDB可以反汇编内核函数，查看全局变量值。

相对于用户态的gdb调试，缺的主要是启动过程调试、函数单步跟踪的调试了。

统信软件
UNIONTECH

2.1.2 BPF实现架构



信软件
ONTECH

2.1.3 BPF实现原理

BPF基于kprobe原理实现，它的本质就是在被跟踪函数的首地址，动态插入一条触发异常的指令，然后基于异常处理流程(也是中断的一种)来调用el1_dbg, 通过异常向量号调用到异常处理函数，最终调用到kprobe处理函数。

对于arm，插入的异常处理指令是brk, 对于x86插入的异常处理指令是ud2.

异常也是中断的一种，都有中断上下文的切换，中断上下文借用当前进程的上下文，当前进程上下文可以保存到当前进程的task_struct中，但是如果中断处理函数再次发生上下文切换就出问题了，因为中断处理程序本身的寄存器信息已经没有地方保存了，因为中断本身没有自己的task_struct。 kprobe处理函数基于异常也就是基于中断机制来处理，所以kprobe处理函数中不能有等待相关的调用触发，因为等待会触发生上下文切换。

BPF只是一个工具的名称，libbcc.so动态库信赖系统调用perf_event_open，完成依赖内存的kprobe_register完成跟踪点的设置。 一旦kprobe_register完成注册，后面对这个跟踪函数的调用注册的内核回调函数就会调用了。对于BPF, 如果跟踪进入时的调用，内核注册的回调函数是kprobe_dispatcher, 如果跟踪的返回值，内核注册的回调函数是pre_handler_kretprobe。

本节将详细介绍如下几个知识点：异常处理指令是如何被注入进来的，内核注册的回调函数如何被调用的，回调函数本身做了什么。

统信软件 UNIONTECH

2.1.4 异常指令的注入

以根据do_mmap的调用为例，当没有跟踪它时，do_mmap函数开始的几行汇编代码如下:

```
(gdb) disas do_mmap
Dump of assembler code for function do_mmap:
  0xffff000008247910 <+0>:    stp    x29, x30, [sp, #-112]!
  0xffff000008247914 <+4>:    mov    x29, sp
  0xffff000008247918 <+8>:    stp    x21, x22, [sp, #32]
```

开始跟踪:

```
trace-bpfcc -t do_mmap
TIME    PID    TID    COMM        FUNC
```

此时的汇编代码如下:

```
(gdb) disas do_mmap
Dump of assembler code for function do_mmap:
  0xffff000008247910 <+0>:    brk    #0x4
  0xffff000008247914 <+4>:    mov    x29, sp
  0xffff000008247918 <+8>:    stp    x21, x22, [sp, #32]
```

统信软件 UNIONTECH

2.1.4.2 异常指令的注入

```
(gdb) disas do_mmap
Dump of assembler code for function do_mmap:
   0xffff000008247910 <+0>:     stp    x29, x30, [sp, #-112]!
   0xffff000008247914 <+4>:     mov    x29, sp
   0xffff000008247918 <+8>:     stp    x21, x22, [sp, #32]
```

开始跟踪:

```
trace-bpfcc -t do_mmap
TIME    PID    TID    COMM        FUNC
```

此时的汇编代码如下:

```
(gdb) disas do_mmap
Dump of assembler code for function do_mmap:
   0xffff000008247910 <+0>:     brk    #0x4
   0xffff000008247914 <+4>:     mov    x29, sp
   0xffff000008247918 <+8>:     stp    x21, x22, [sp, #32]
```

do_mmap首地址处的第一条指令补临时替换成了brk #04指令

UNIONTECH

2.1.4.3 brk指令注入代码

从register_kprobe出发，最终调用arch_arm_kprobe完成注册

register_kprobe --> arm_kprobe --> arch_arm_kprobe

arch_arm_kprobe的代码如下：

```
/* arm kprobe: install breakpoint in text */
void __kprobes arch_arm_kprobe(struct kprobe *p)
{
    patch_text(p->addr, BRK64_OPCODE_KPROBES);
}
#define BRK64_OPCODE_KPROBES    (AARCH64_BREAK_MON | (BRK64_ESR_KPROBES << 5))
#define BRK64_ESR_KPROBES   0x0004
#define AARCH64_BREAK_MON   0xd4200000
```

BRK64_OPCODE_KPROBES的值就是0xd4200080，它是brk #04指令的机器码

p->addr是0xffff000008247910

总结：

最后向do_mmap的首地址0xffff000008247910写了0xd4200080这个机器码。

统信软件 UNIONTECH

2.1.5 基于BPF本身查看注册后的值

a)跟踪传给register_kprobe的参数

```
sudo trace-bpfcc -tK 'register_kprobe(struct kprobe *p) "p=0x%lx",p'
```

b)开始跟踪do_mmap

```
sudo trace-bpfcc -t do_mmap -p 11
```

c)此时的打印结果如下:

```
TIME     PID     TID     COMM          FUNC          -
6.384717 55310   55310   trace-bpfcc   register_kprobe  p=0xffffa0276389c210
```

d)gdb查看p的值

```
p *(struct kprobe*)0xffffa0276389c210
$2 = {hlist = {next = 0x0, pprev = 0xffff000009a2a8a8 <kprobe_table+232>}, list = {
   next = 0xffffa0276389c220, prev = 0xffffa0276389c220}, nmissed = 0,
  addr = 0xffff000008247910 <do_mmap>, symbol_name = 0xffffa02763d93480 "do_mmap",
  offset = 0, pre_handler = 0xffff0000081b6b90 <kprobe_dispatcher>, post_handler = 0x0,
  fault_handler = 0x0, opcode = 2847505405, ainsn = {api = {insn = 0xffff00000ab1000c,
    pstate_cc = 0x0, handler = 0x0, restore = 18446462598869448980}}, flags = 0}
```

统信软件 UNIONTECH

2.1.5.2 基于BPF本身查看注册后的值

p->addr=0xffff000008247910,　　它就是do_mmap的首地址

p->symbol_name="do_mmap"，它就是需要跟踪的函数

pre_handler=kprobe_dispatcher，brk指令最后保证kprobe_dispatcher函数被调用

opcode=2847505405=0xa9b97bfd，它是谁呢？为什么不是前面看到的0xd4200080呢？

ainsn = {api = {insn = 0xffff00000ab1000c，它的值是0xa9b97bfd

(gdb) x 0xffff00000ab1000c

0xffff00000ab1000c:　0xa9b97bfd

(gdb) x 0xffff000008247910

0xffff000008247910 <do_mmap>:　0xd4200080　可以确定do_mmap首地址的值是0xd4200080，没有疑问

现在基于gdb确认一下0xd4200080和0xa9b97bfd机器码对应的汇编指令。

统信软件
UNIONTECH

2.1.6 gdb确认机器码汇编指令

```
int main(int argc, char** argv) {
	int *myopcode=malloc(64);
	myopcode[0]=0xa9b97bfd;
	myopcode[1]=0xd4200080;
	return 0;
}gdb) p myopcode
$1 = (int *) 0x420260
(gdb) disas 0x420260,0x420268
Dump of assembler code from 0x420260 to 0x420268:
   0x0000000000420260:  stp    x29, x30, [sp, #-112]!   //0xa9b97bfd就是以前的指令
   0x0000000000420264:  brk    #0x4                      //0xd4200080就是替换后的指令
```

总结：

a)一个被跟踪的函数的首地址被替换成了brk #04

b)被替换的指令被保存了，存在了两个地方,第一个地方是p->opcode, 在arch_prepare_kprobe函数中完成.

第二个地方在p->ainsn.api.insn,也在arch_prepare_kprobe函数中完成。

c)可以借助gdb在用户态反汇编机器码方便我们调试

统信软件 UNIONTECH

2.1.7 异常处理函数的注册

el1_dbg --> do_debug_exception

--> const struct fault_info *inf = debug_fault_info + DBG_ESR_EVT(esr)

--> inf->fn(addr_if_watchpoint, esr, regs)

```
static int __init debug_traps_init(void)
{
    hook_debug_fault_code(DBG_ESR_EVT_HWSS, single_step_handler, SIGTRAP,
                TRAP_TRACE, "single-step handler");
    hook_debug_fault_code(DBG_ESR_EVT_BRK, brk_handler, SIGTRAP,
                TRAP_BRKPT, "ptrace BRK handler");
    return 0;
}
#define DBG_ESR_EVT_HWSS    0x1   //用于设置处理函数
#define DBG_ESR_EVT_BRK     0x6    //设置brk处理函数
```

总结：

hook_debug_fault_code将single_step_handler和brk_handler两个函数注册到了debug_fault_info中。

即debug_fault_info[1]将取出single_step_handler来执行，debug_fault_info[6]将取出brk_handler来执行

统信软件 UNIONTECH

2.1.8 先调用brk_handler

single_step_handler和brk_handler两个函数已经无法通过BPF来跟踪了，通过增加打印发现，brk_handler和single_step_handler两个函数依次被调用。

也就是说brk #04指令首先触发的是DBG_ESR_EVT_BRK=6事件，然后触发了DBG_ESR_EVT_HWSS=1事件

```
[ 1737.205807] brk_handler invoked pid=2571 addr=0xfffcb3c73f80 esr=0xf2000004[2696]
[ 1737.205809] CPU: 54 PID: 2571 Comm: mmap Tainted: G        W        4.19.0-arm64-server
-unixbench #180
[ 1737.205809] Hardware name: Huawei S920X00K/BC82AMDDIA, BIOS 1.21 02/02/2021
[ 1737.205810] Call trace:
[ 1737.205811]  dump_backtrace+0x0/0x190
[ 1737.205812]  show_stack+0x14/0x20
[ 1737.205813]  dump_stack+0xa8/0xcc
[ 1737.205815]  brk_handler+0x3c/0xe0
[ 1737.205816]  do_debug_exception+0x84/0x158
[ 1737.205817]  el1_dbg+0x18/0x78
[ 1737.205819]  do_mmap+0x0/0x488
[ 1737.205820]  ksys_mmap_pgoff+0x1bc/0x230
[ 1737.205821]  __arm64_sys_mmap+0x28/0x38
[ 1737.205823]  el0_svc_common+0x90/0x160
[ 1737.205824]  el0_svc_handler+0x9c/0xa8
[ 1737.205825]  el0_svc+0x8/0xc
```

1)0xf2000004>>27=0x1e

0x1e & 7=6, 所以调用了brk_handler

2)调用了brk_handler

统信软件 UNIONTECH

2.1.9 再调用signle_step_handler

brk #04指令然后触发的是DBG_ESR_EVT_HWSS=1事件

```
49985.370376] single_step_handler invoked pid=5969 addr=0xfffec32a3f80 esr=0xcf000022[7859]
49985.370377] CPU: 19 PID: 5969 Comm: mmap Not tainted 4.19.0-arm64-server-unixbench #181
49985.370378] Hardware name: Huawei S920X00K/BC82AMDDIA, BIOS 1.21 02/02/2021
49985.370378] Call trace:
49985.370379]  dump_backtrace+0x0/0x190
49985.370380]  show_stack+0x14/0x20
49985.370381]  dump_stack+0xa8/0xcc
49985.370382]  single_step_handler+0x3c/0x138
49985.370383]  do_debug_exception+0x84/0x158
49985.370384]  el1_dbg+0x18/0x78
49985.370385]  0xffff00000a49000c
49985.370387]  ksys_mmap_pgoff+0x1bc/0x230
49985.370387]  __arm64_sys_mmap+0x28/0x38
49985.370389]  el0_svc_common+0x90/0x160
49985.370390]  el0_svc_handler+0x9c/0xa8
49985.370391]  el0_svc+0x8/0xc
```

0xcf000027 >> 27=0x19

0x19 & 7 = 1

所以执行single_step_handler

统信软件 UNIONTECH

2.1.10 kprobe_dispatcher最终处理函数的调用

a)首先基于gdb确认调用的是kprobe_perf_func函数

```
extern int g_debug_kprobe;
static int kprobe_dispatcher(struct kprobe *kp, struct pt_regs *regs)
{
    struct trace_kprobe *tk = container_of(kp, struct trace_kprobe, rp.kp);
    int ret = 0;

    raw_cpu_inc(*tk->nhit);
    if (g_debug_kprobe & 4) {
        printk("%s invoked tp.flags=0x%lx pid=%d tk=0x%lx\n", __FUNCTION__, tk->tp.flags, current->pid, tk);
    }
    if (tk->tp.flags & TP_FLAG_TRACE)
        kprobe_trace_func(tk, regs);
#ifdef CONFIG_PERF_EVENTS
    if (tk->tp.flags & TP_FLAG_PROFILE)
        ret = kprobe_perf_func(tk, regs);
#endif
    return ret;
}
```

1)假设bfp获取到的kp为0xffffa027c9f6c010,那么tk的地址向前偏移16B

p ((struct trace_kprobe*)(0xffffa027c9f6c010-16))->tp.flags就可以知道走哪个分支了

2)打印出来是6, 走的是下面这个分支

统信软件 UNIONTECH

2.1.11 寻找最终调用函数

kprobe_perf_func函数进一步调用trace_call_bpf

最后基于下面BPF_PROG_RUN_ARRAY_CHECK完成对bpf程序的调用

```
struct trace_kprobe *tk=container_of(kp, struct trace_kprobe, rp.kp);
struct trace_event_call *call = &tk->tp.call;
ret = BPF_PROG_RUN_ARRAY_CHECK(call->prog_array, ctx, BPF_PROG_RUN)
```

已知bpf获取到的struct kprobe地址为：

```
sudo trace-bpfcc -t 'register_kprobe(struct kprobe *p) "p=0x%lx",p'
499.1753 5617    5617    trace-bpfcc     register_kprobe  p=0xffffa027c9d7fa10
```

将宏展开理解后，实现上最后调用的代码就是

```
((struct trace_kprobe*)(0xffffa027c9d7fa10-16))->tp.call.prog_array[0].items[0].prog->bpf_func
```

下面基于反汇编看看它是什么样的代码，gdb中打印这个代码的首地址，然后反汇编来查看理解。

统信软件 UNIONTECH

2.1.12 寻找最终调用函数

```
p ((struct trace_kprobe*)(0xffffa027c9d7fa10-16))->tp.call.prog_array[0].items[0].prog->bpf_func
$17 = (unsigned int (*)(const void *, const struct bpf_insn *)) 0xffff000003d596ac
```

反汇编一段代码块

```
disas 0xffff000003d596ac,0xffff000003d597ac
```

```
0xffff000003d596d0:  mov     x10, #0xffff000000000000       // #-281474976710656
0xffff000003d596d4:  movk    x10, #0x81d, lsl #16
0xffff000003d596d8:  movk    x10, #0xda0
0xffff000003d596dc:  blr     x10
```

第一个调用的函数

```
x (0xffff000000000000 + (0x81d<<16)+0xda0)
0xffff0000081d0da0 <bpf_get_current_pid_tgid>:
```

第一个函数是获取tgid

```
0xffff000003d59718:  mov     x10, #0xffff000000000000       // #-281474976710656
0xffff000003d5971c:  movk    x10, #0x81d, lsl #16
0xffff000003d59720:  movk    x10, #0xdf0
0xffff000003d59724:  blr     x10
```

第二个调用的函数

```
x (0xffff000000000000 + (0x81d<<16)+0xdf0)
0xffff0000081d0df0 <bpf_ktime_get_ns>:
```

第一个函数是获取时间

```
0xffff000003d59754:  mov     x1, #0x10                      // #16
0xffff000003d59758:  mov     x10, #0xffff000000000000       // #-281474976710656
0xffff000003d5975c:  movk    x10, #0x81d, lsl #16
0xffff000003d59760:  movk    x10, #0xe70
0xffff000003d59764:  blr     x10
```

第三个调用的函数

```
x (0xffff000000000000 + (0x81d<<16)+0xe70)
0xffff0000081d0e70 <bpf_get_current_comm>:
```

第一个函数是获取进程名

统信软件
UNIONTECH

2.1.13 寻找最终调用函数

```
0xffff000003d59788:  mov     w2, #0xffffffff                  // #-1
0xffff000003d5978c:  mov     x4, #0x20                        // #32
0xffff000003d59790:  mov     x10, #0xffff000000000000         // #-281474976710656
0xffff000003d59794:  movk    x10, #0x81b, lsl #16
0xffff000003d59798:  movk    x10, #0x4d20
0xffff000003d5979c:  blr     x10
0xffff000003d597a0:  add     x7, x0, #0x0
0xffff000003d597a4:  mov     x7, #0x0                         // #0
0xffff000003d597a8:  add     sp, sp, #0x20
0xffff000003d597ac:  ldp     x25, x26, [sp], #16
0xffff000003d597b0:  ldp     x21, x22, [sp], #16
0xffff000003d597b4:  ldp     x19, x20, [sp], #16
0xffff000003d597b8:  ldp     x29, x30, [sp], #16
0xffff000003d597bc:  add     x0, x7, #0x0
0xffff000003d597c0:  ret
```

第四个函数，也是最后一个调用函数

x (0xffff000000000000 + (0x81b<<16)+0x4d20)

0xffff0000081b4d20 <bpf_perf_event_output>:

第四个函数是将结果输出来

统信软件 UNIONTECH

2.1.14 这些BPF函数在哪儿

1)bpf_get_current_pid_tgid

kernel/bpf/helpers.c

```
BPF_CALL_0(bpf_get_current_pid_tgid)
{
    struct task_struct *task = current;
    if (unlikely(!task))
        return -EINVAL;
    return (u64) task->tgid << 32 | task->pid;
}//实现很简单，就是获取当前进程的pid和tgid
```

2)bpf_get_current_pid_tgid

kernel/bpf/helpers.c

```
BPF_CALL_0(bpf_ktime_get_ns)
{
    /* NMI safe access to clock monotonic */
    return ktime_get_mono_fast_ns();
}//实现很简单，就是获取当前时间
```

3)bpf_get_current_comm

kernel/bpf/helpers.cBPF_CALL_2(bpf_get_current_comm, char *, buf, u32, size)

```
{
    struct task_struct *task = current;
    strncpy(buf, task->comm, size);
}//实现很简单，就是获取当前进程名字
```

统信软件 UNIONTECH

2.1.15 总结

A)BPF是Berkeley Packet Filter的简称，它最初来源于网络包的过滤，但是现在它已经被实现成了函数的运态跟 踪功能了。

B)BPF在内核的实现依赖kprobe, 依赖kprobe基于异常的代码动态注入机制，在能被跟踪的函数的首个地址注入一条触发异常的指令，如arm上为brk #04, x86上为ud2, arm64上的注入代码就是patch_text(p->addr, BRK64_OPCODE_KPROBES);

C)当运行到被跟踪的函数时，异常指令触发异常，开始进入kprobe的通道。

D)kprobe的异常处理函数brk_handler和single_step_handler依次被执行，brk_handler最终调用的是pre_handler回调函数，即kprobe_dispather函数(如果是kretprobe, 那么调用的是pre_handler_kretprobe函数)；signle_step_handler最终调用的是post_handler回调函数(本文没有介绍这种情况，在ko中写代码时可以显示给post_handler注册回调函数)

E)kprobe_dispatcher最终以回调函数形式调用BPF模块提供的函数：bpf_get_current_pid_tgid、bpf_ktime_get_ns、bpf_get_current_comm、bpf_perf_event_output

统信软件
UNIONTECH

2.2 跟踪内存申请

linux上经常说一个概念，就是内存只有到读写时才会申请内存，现在通过bpf来验证。

```
int writemem(int needwrite)
{
    int len=1024*1024;
    char *psz=malloc(len);
    if (needwrite) {
      memset(psz,0,len);
    }
    return 0;
}
```

```
int main(int argc, char**argv)
{
    int needwrite = 0;
    if (argc > 1) {
        needwrite = atoi(argv[1]);
    }
    printf("pid=%d\n",getpid());
    getchar();
    writemem(needwrite);
    printf("finish test\n");
    getchar();
    return 0;
}
```

统信软件 UNIONTECH

2.2.1 代码理解

linux上经常说一个概念，就是内存只有到读写时才会申请内存，现在通过bpf来验证。

```
int writemem(int needwrite)
{
    int len=1024*1024;
    char *psz=malloc(len);
    if (needwrite) {
      memset(psz,0,len);
    }
    return 0;
}
```

3)申请了1M的空间，C库内部调用mmap申请，所以这儿只是申请地址空间，内核创建VMA

4)后面测试会发现，这儿才会真正去申请内存

```
int main(int argc, char**argv)
{
    int needwrite = 0;
    if (argc > 1) {
        needwrite = atoi(argv[1]);
    }
    printf("pid=%d\n",getpid());
    getchar();
    writemem(needwrite);
    printf("finish test\n");
    getchar();
    return 0;
}
```

1)控制是否写申请的内存

2)构造等待，查看内存申请情况

统信软件 UNIONTECH

2.2.2 跟踪内存申请

A)首先看一下只申请内存，不写数据的情况，

此时needwrite=0

```
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$ ./writemem
pid=31592

finish test
```

1)没有传入参数，此时只调用了malloc

```
uos@uos-PC:~/zhoupeng/train/method2$ sudo trace-bpfcc -tK __alloc_pages_nodemask -p 31592
TIME     PID     TID     COMM          FUNC
1.709517 31592   31592   writemem      __alloc_pages_nodemask
        __alloc_pages_nodemask+0x0 [kernel]
        __handle_mm_fault+0x878 [kernel]
        handle_mm_fault+0xec [kernel]
        do_page_fault+0x168 [kernel]
        do_translation_fault+0x58 [kernel]
        do_mem_abort+0x3c [kernel]
        el0_da+0x20 [kernel]
```

2)-t表示打印时间，K表示打印内核堆栈

3) -p指定了需要跟踪的进程

4)用户态触发的写内存

trace-bpfcc是基于python写bpf跟踪工具，内部基于kprobe实现

通过下面的命令来安装：sudo apt install bpfcc-tools

调用malloc只申请了一次物理页内存，但malloc本身申请了1M, 不可能是这个1M对应的内存，了解c库内存管理代码就可以知道，申请内存时c库本身需要申请需要管理的内存，这个内存就是c库内部自己申请的。

统信软件 UNIONTECH

2.2.3 跟踪内存申请

B)查看申请内存的大小

```
sudo trace-bpfcc -tK '__alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order)
"order=%d",order' -p 32940
TIME     PID    TID    COMM          FUNC           -
2.568359 32940  32940  writemem      __alloc_pages_nodemask order=0
        __alloc_pages_nodemask+0x0 [kernel]
        __handle_mm_fault+0x878 [kernel]
        handle_mm_fault+0xec [kernel]
        do_page_fault+0x168 [kernel]
        do_translation_fault+0x58 [kernel]
        do_mem_abort+0x3c [kernel]
        el0_da+0x20 [kernel]
```

2）打印order

1)想将order打印出来，因此需要提供order及order之前的参数

3)order=0, 所以只申请一个物理页

统信软件 UNIONTECH

2.2.4 跟踪内存申请

C)查看写内存时的物理内存申请情况

```
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$ ./writemem 1
pid=33875

finish test
```

needwrite=1,因此会调用memset写申请的内存

```
uos@uos-PC:~/zhoupeng/train/method2$ sudo trace-bpfcc -t '__alloc_pages_nodemask(gfp_t gfp_mask, unsigned int ord
er) "order=%d",order' -p 33875
TIME     PID     TID     COMM         FUNC            -
2.200159 33875   33875   writemem     __alloc_pages_nodemask order=0
2.200176 33875   33875   writemem     __alloc_pages_nodemask order=0
2.200183 33875   33875   writemem     __alloc_pages_nodemask order=0
2.200189 33875   33875   writemem     __alloc_pages_nodemask order=0
2.200196 33875   33875   writemem     __alloc_pages_nodemask order=0
2.200202 33875   33875   writemem     __alloc_pages_nodemask order=0
2.200208 33875   33875   writemem     __alloc_pages_nodemask order=0
2.200214 33875   33875   writemem     __alloc_pages_nodemask order=0
2.200220 33875   33875   writemem     __alloc_pages_nodemask order=0
2.200226 33875   33875   writemem     __alloc_pages_nodemask order=0
2.200233 33875   33875   writemem     __alloc_pages_nodemask order=0
2.200239 33875   33875   writemem     __alloc_pages_nodemask order=0
2.200245 33875   33875   writemem     __alloc_pages_nodemask order=0
2.200251 33875   33875   writemem     __alloc_pages_nodemask order=0
2.200257 33875   33875   writemem     __alloc_pages_nodemask order=0
2.200265 33875   33875   writemem     __alloc_pages_nodemask order=0
2.200271 33875   33875   writemem     __alloc_pages_nodemask order=0
```

2) 16页，16*64K=1M
刚好是要写的内存空间

memset触发了物理内存的申请，写了1M, 也刚好申请了16*64K=1M的内存

统信软件 UNIONTECH

2.3 跟踪io流程

A)dd缓存方式写文件

```
Starting program: /usr/bin/dd if=/dev/zero of=1G.txt bs=1G count=1

Breakpoint 1, __libc_open64 (file=0xfffffffe1d7 "/dev/zero", oflag=0)
    at ../sysdeps/unix/sysv/linux/open64.c:37
37      ../sysdeps/unix/sysv/linux/open64.c: 没有那个文件或目录.
(gdb) she ps -ef|grep /usr/bin/dd|grep if
uos      34708 34606  0 20:09 pts/6    00:00:00 /usr/bin/dd if=/dev/zero of=1G.txt
(gdb) dis
(gdb) c
Continuing.
记录了1+0 的读入
记录了1+0 的写出
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 0.299729 s, 3.6 GB/s
[Inferior 1 (process 34708) exited normally]
(gdb)
```

1)dd缓存方式写文件

3) 获取需要跟踪的进程，为34708

```
^C
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$
uos@uos-PC:~/zhoupeng/train/method2$ sudo trace-bpfcc -t submit_bio -p 34708
TIME     PID     TID     COMM         FUNC
```

2)跟踪submit_bio, 如果提交数据到磁盘，就会调用此函数

通过对bpf跟踪submit_bio函数发现没有向磁盘提交数据，全部是写缓存

统信软件
UNIONTECH

2.3.2 跟踪io流程

B)dd direct方式写文件

```
(gdb) r
Starting program: /usr/bin/dd if=/dev/zero of=1G.txt bs=1G count=1 oflag=direct

Breakpoint 1, __libc_open64 (file=0xfffffffe1ca "/dev/zero", oflag=0)
    at ../sysdeps/unix/sysv/linux/open64.c:37
37      ../sysdeps/unix/sysv/linux/open64.c: 没有那个文件或目录.
(gdb) set args if=/dev/zero of=1G.txt bs=1G count=1 oflag=direct
(gdb) she ps -ef|grep /usr/bin/dd|grep if
uos      37056 34606  0 20:33 pts/6    00:00:00 /usr/bin/dd if=/dev/zero of=1G.txt
(gdb) dis
(gdb) c
Continuing.
记录了1+0 的读入
记录了1+0 的写出
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 0.27168 s, 4.0 GB/s
[Inferior 1 (process 37056) exited normally]
(gdb)
```

1)direct方式写文件

2)37056是需要被跟踪的进程

```
4.253539 37056   37056   dd        submit_bio
4.256102 37056   37056   dd        submit_bio
4.258691 37056   37056   dd        submit_bio
4.261262 37056   37056   dd        submit_bio
4.263873 37056   37056   dd        submit_bio
4.266445 37056   37056   dd        submit_bio
4.269011 37056   37056   dd        submit_bio
4.271608 37056   37056   dd        submit_bio
4.274210 37056   37056   dd        submit_bio
4.276777 37056   37056   dd        submit_bio
4.279362 37056   37056   dd        submit_bio
4.281945 37056   37056   dd        submit_bio
4.284554 37056   37056   dd        submit_bio
4.287110 37056   37056   dd        submit_bio
4.289697 37056   37056   dd        submit_bio
4.292278 37056   37056   dd        submit_bio
```

3)每次提交16M,
总共调用64次,
16*64=1G

direct方式写数据时就会调用submit_bio, 将数据提交到io queue

统信软件 UNIONTECH

2.4 获取回调函数名字

以IO为例，submit_bio调用generic_make_request, 该函数会调用q->make_request_fn将bio发送取出

A)基于bpf和kallsysm信息来获取q->make_request_fn是谁

```
uos@uos-PC:~$ sudo trace-bpfcc -t 'submit_bio(struct bio *bio) "func=0x%lx",bio->b
i_disk->queue->make_request_fn' -p 3284
TIME      PID    TID    COMM          FUNC          -
5.133315  3284   3284   dd            submit_bio       func=0xffff000008478e18
5.133475  3284   3284   dd            submit_bio       func=0xffff000008478e18
5.133624  3284   3284   dd            submit_bio       func=0xffff000008478e18
5.133753  3284   3284   dd            submit_bio       func=0xffff000008478e18
5.133910  3284   3284   dd            submit_bio       func=0xffff000008478e18
5.133993  3284   3284   dd            submit_bio       func=0xffff000008478e18
5.134080  3284   3284   dd            submit_bio       func=0xffff000008478e18
5.134152  3284   3284   dd            submit_bio       func=0xffff000008478e18
5.134250  3284   3284   dd            submit_bio       func=0xffff000008478e18
5.135612  3284   3284   dd            submit_bio       func=0xffff000008478e18
5.138127  3284   3284   dd            submit_bio       func=0xffff000008478e18
5.140710  3284   3284   dd            submit_bio       func=0xffff000008478e18
5.143329  3284   3284   dd            submit_bio       func=0xffff000008478e18
5.145878  3284   3284   dd            submit_bio       func=0xffff000008478e18
5.148462  3284   3284   dd            submit_bio       func=0xffff000008478e18
5.151043  3284   3284   dd            submit_bio       func=0xffff000008478e18
```

2)把这个函数地址打印出来

3)这个是函数的地址

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /usr/bin/dd...(no debugging symbols found)...done.
(gdb) set args if=/dev/zero of=1G.txg bs=1G count=1 oflag=direct
(gdb) b open
Breakpoint 1 at 0x1e24
(gdb) r
Starting program: /usr/bin/dd if=/dev/zero of=1G.txg bs=1G count=1 oflag=direct

Breakpoint 1, __libc_open64 (file=0xfffffffe1cd "/dev/zero", oflag=0)
    at ../sysdeps/unix/sysv/linux/open64.c:37
37      ../sysdeps/unix/sysv/linux/open64.c: 没有那个文件或目录.
(gdb) she ps -ef|grep /usr/bin/dd|grep if
uos       3284  3275  0 22:42 pts/3    00:00:00 /usr/bin/dd if=/dev/zero of=1G.txg
(gdb) dis
```

4)direct方式才会submit_bio

```
2455                enter_succeeded = false;
2456        }
2457
2458        if (enter_succeeded) {
2459            struct bio_list lower, same;
2460
2461            /* Create a fresh bio_list for all subordinate requests */
2462            bio_list_on_stack[1] = bio_list_on_stack[0];
2463            bio_list_init(&bio_list_on_stack[0]);
2464            ret = q->make_request_fn(q, bio);
2465
2466            /* sort new bios into those for a lower level
2467             * and those for the same level
2468             */
2469            bio_list_init(&lower);
2470            bio_list_init(&same);
2471            while ((bio = bio_list_pop(&bio_list_on_stack[0])) != NULL
)
2472                if (q == bio->bi_disk->queue)
2473                    bio_list_add(&same, bio);
2474                else
2475                    bio_list_add(&lower, bio);
2476            /* now assemble so we handle the lowest level first */
2477            bio_list_merge(&bio_list_on_stack[0], &lower);
2478            bio_list_merge(&bio_list_on_stack[0], &same);
2479            bio_list_merge(&bio_list_on_stack[0], &bio_list_on_stack[1
]);
```

1)这个回调函数是谁呢?

```
ysj
yzylog
zhoupeng
uos@uos-PC:~$ sudo cat /proc/kallsyms |grep ffff000008478e18
ffff000008478e18 t blk_queue_bio
uos@uos-PC:~$
```

5)根据打印出来的地址查找函数，原来是blk_queue_bio，单队列情况下的bio发送函数

件 UNIONTECH

2.4.2 获取回调函数名字

B)详细步骤如下：

i)struct request_queue *q = bio->bi_disk->queue

所以q->make_request_fn就是bio->bi_disk->queue->make_request_fn, 因此bfp中打印这个变量就可以了

因此bpf跟踪代码如下:

```
sudo trace-bpfcc -t 'submit_bio(struct bio *bio) "func=0x%lx",bio->bi_disk->queue->make_request_fn'
```

ii)gdb启动，对open设置断点，这样运行之前可以知道dd进程的pid

iii)运行，发现func=0xffff000008478e18

iv)查找kallsyms知道此地址对应的函数

```
sudo cat /proc/kallsyms |grep ffff000008478e18
ffff000008478e18 t blk_queue_bio
```

blk_queue_bio是单队列的bio发送函数：./drivers/mtd/mtd_blkdevs.c

blk_init_queue --> blk_init_queue_node --> blk_init_allocated_queue -->

blk_queue_make_request(q, blk_queue_bio);

统信软件 UNIONTECH

2.5 基于bpftrace写性能统计代码

A)direct同写1G文件，为什么iozone的性能几乎是dd的两倍？dd耗时在哪儿了？如何快速确认？

dd命令如下：

dd if=/dev/zero of=1G.txt bs=16M count=64 oflag=direct

测试结果如下：

0.279126 s, 3.8 GB/s

iozone命令如下：

./iozone -+n -s 1g -r 16m -i 0 -I

测试结果为：6112644KB/s=5.83GB/s

统信软件
UNIONTECH

2.5.2 基于bpftrace写性能统计代码

B)基于 trace-bpfcc熟悉dd流程

先确认dd、iozone调用了read/write函数情况，如下实现：

```
gdb /usr/bin//dd
set args if=/dev/zero of=1G.txt bs=16M count=64 oflag=direct
b open
```

停下来后得知dd进程pid为9609

```
sudo trace-bpfcc -t vfs_read vfs_write -p 9609
TIME     PID    TID    COMM        FUNC
3.997389 9609   9609   dd          vfs_read
3.999713 9609   9609   dd          vfs_write
...
4.011275 9609   9609   dd          vfs_read
4.012946 9609   9609   dd          vfs_write
...
```

vfs_read, vfs_write读写次数各64次，所以dd是一边读、一边写的.

统信软件 UNIONTECH

2.5.3 基于bpftrace写性能统计代码

C)基于 trace-bpfcc熟悉iozone流程

```
./iozone -+n -s 1g -r 16m -i 0 -I
```

iozone进程pid=12040

```
sudo trace-bpfcc -t vfs_read vfs_write -p 12040
TIME     PID     TID     COMM          FUNC
2.434148 10240   10240   iozone        vfs_write
2.434166 10240   10240   iozone        vfs_write
2.434173 10240   10240   iozone        vfs_write
2.434178 10240   10240   iozone        vfs_write
...
```

vfs_write全部为写，所以iozone仅仅是将内存中的数据写入输出文件，估计这个是iozone比dd快的根本原因，继续基于bpf写性能统计代码

统信软件 UNIONTECH

2.5.4 基于bpftrace写性能统计代码

D)基于bpf写性能统计代码

```
#!/usr/bin/bpftrace
BEGIN
{
	printf("begin stat io pid=%d\n",$1);
}
kprobe:vfs_read
/pid==$1/
{
	@read_nsecs=nsecs;
	@read_nrs = @read_nrs + 1;
}
kretprobe:vfs_read
/@read_nsecs !=0/
{
  if (pid==$1)
  {
	@total_read_nsecs = @total_read_nsecs + (nsecs - @read_nsecs);
  }
}
kprobe:vfs_write
/pid==$1/
{
  @write_nsecs=nsecs;
  @write_nrs = @write_nrs + 1;
}
kretprobe:vfs_write
/@write_nsecs !=0/
{
	if (pid==$1)
	{
		@total_write_nsecs = @total_write_nsecs + (nsecs -
@write_nsecs);
	}
}
```

统信软件 UNIONTECH

2.5.5 基于bpftrace写性能统计代码

D)基于bpf写性能统计代码

```
END
{
    printf("read_nrs=%d; read_ms=%ld\n", @read_nrs, @total_read_nsecs/1000/1000);
    printf("write_nrs=%d; write_ms=%ld\n", @write_nrs, @total_write_nsecs/1000/1000);
    clear(@read_nsecs);
    clear(@read_nrs);
    clear(@total_read_nsecs);
    clear(@write_nsecs);
    clear(@write_nrs);
    clear(@total_write_nsecs);
}
```

统信软件
UNIONTECH

2.5.6 基于bpftrace写性能统计代码

E)bpf代码解释

d)基于bpf写性能统计代码

```
#!/usr/bin/bpftrace
BEGIN
{
    printf("begin stat io pid=%d\n",$1);
}
kprobe:vfs_read
/pid==$1/
{
    @read_nsecs=nsecs;
    @read_nrs = @read_nrs + 1;
}
kretprobe:vfs_read
/@read_nsecs !=0/
{
  if (pid==$1)
  {
    @total_read_nsecs = @total_read_nsecs + (nsecs - @read_nsecs);
  }
}
```

1)只统计关心的进程

2）记录进入read函数的时间

3)记录read函数的运行次数

4)确保本次统计的read进入时间已经捕捉到了

5)也只统计关心的进程

6）统计read函数运行的总时间

```
kprobe:vfs_write
/pid==$1/
{
  @write_nsecs=nsecs;
  @write_nrs = @write_nrs + 1;
}
kretprobe:vfs_write
/@write_nsecs !=0/
{
    if (pid==$1)
    {
        @total_write_nsecs = @total_write_nsecs + (nsecs -
@write_nsecs);
    }
}
```

7)统计write函数运行的总时间

统信软件 UNIONTECH

统信软件 UNIONTECH

2.5.7 基于bpftrace写性能统计代码

F)性能统计代码验证iozone

./iozone -+n -s 1g -r 16m -i 0 -I

测试结果为：6110478KB/s=5.83GB/s

已知iozone进程的pid为11617

sudo ./statio-all.kp 11617

Attaching 6 probes...

begin stat io pid=11617

read_nrs=2; read_ms=

write_nrs=114; write_ms=171

1.0GB/0.171s=5.84GB/s

结论：

iozone花的时间基本上就是write函数花的时间

统信软件
UNIONTECH

2.5.8 基于bpftrace写性能统计代码

G)分性能统计代码验证dd

/usr/bin/dd if=/dev/zero of=1G.txt bs=16M count=64 oflag=direct

测试结果为：0.279561 s, 3.8 GB/s

已知dd进程的pid为51381

sudo ./statio-all.kp 51381

Attaching 6 probes...

begin stat io pid=51381

read_nrs=66; read_ms=107

write_nrs=67; write_ms=172

结论：

总时间为0.107+0.172=0.279s，和dd测试时间一样

dd花的时间基本上就是read、write两个函数花的时间

统信软件
UNIONTECH

2.6 基于bpftrace分析性能

A)继续上面的iozone来分析，看看vfs_write函数的时间主要花在哪个函数上了.

由于io读写最终会调用submit_bio函数，因此基于二分法思想先确认submit_bio函数花的时间，确定它是同步还是异步。

对submit_bio增加如下调试代码：

```
kprobe:submit_bio
/pid==$1/
{
    @start_nsecs=nsecs;
    @submit_nrs = @submit_nrs + 1;
    if (@start==0) {
        @start = nsecs;
    }
}
```

```
kretprobe:submit_bio
/@start_nsecs !=0/
{           @total_nsecs = @total_ns
    if (pid==$1)
    {
ecs + (nsecs - @start_nsecs);
        @end = nsecs;
    }
}
```

统信软件 UNIONTECH

2.6.2 基于bpftrace分析性能

B)统计性能

./iozone -+n -s 1g -r 16m -i 0 -I

submit_nrs=72; submit_ms=5

总耗时172ms, 但submit_bio本身才花了5ms, 继续分析时间花在哪儿了

获取调用submit_bio的堆栈

sudo trace-bpfcc -tK submit_bio -p 35383

统信软件
UNIONTECH

2.6.3 基于bpftrace分析性能

B)获取调用submit_bio的堆栈

sudo trace-bpfcc -tK submit_bio -p 35383

```
submit_bio+0x0 [kernel]
iomap_dio_actor+0x144 [kernel]
iomap_apply+0x9c [kernel]
iomap_dio_rw+0x21c [kernel]
xfs_file_dio_aio_write+0x118 [kernel]
xfs_file_write_iter+0xc0 [kernel]
__vfs_write+0xf4 [kernel]
vfs_write+0xa4 [kernel]
ksys_write+0x4c [kernel]
__arm64_sys_write+0x18 [kernel]
el0_svc_common+0x90 [kernel]
el0_svc_handler+0x9c [kernel]
el0_svc+0x8 [kernel]
```

由于vfs_write耗时172ms, 而submit_bio耗时只有5ms, 所以耗时更长的函数肯定在上面的堆栈的某个函数中。在还没有理解代码的情况下，可以先基于二分法思想统计上面的iomap_dio_rw函数

统信软件 UNIONTECH

2.6.4 基于bpftrace分析性能

C)统计iomap_dio_rw函数

submit_nrs=64; submit_ms=171

继续看iomap_dio_rw的实现

```
blk_start_plug
do {
    iomap_apply
}
blk_finish_plug
dio->wait_for_completion = wait_for_completion;
if (!(iocb->ki_flags & IOCB_HIPRI)) {
     io_schedule();
}
__set_current_state(TASK_RUNNING);
iomap_dio_complete
```

通过这个函数架构基本可以猜测blk_start_plug和blk_finish_plug之间的只是发送，

iomap_dio_complete需要等待结果,耗时很有可能发生在io_schedule的调用

统信软件 UNIONTECH

2.7 bpf+gdb配合调试

bpf比较方便打印输入参数和输出值，对于全局变量及current的信息打印比较麻烦，或者说基本不可行。bpftrace提供了curtask这个变量，这个变量相当于内核中的current全局变量，但是很难写代码将current内部变量的值打印出来。

由于时间和精力有限，我这边没有成功，但是找到了一个借助gdb配合将当前进程的详细信息打印出来的方法。

统信软件
UNIONTECH

2.7.2 分析场景

```
unsigned long thp_get_unmapped_area(struct file *filp, unsigned long addr,
        unsigned long len, unsigned long pgoff, unsigned long flags)
{
    loff_t off = (loff_t)pgoff << PAGE_SHIFT;

    if (addr)
        goto out;
    if (!IS_DAX(filp->f_mapping->host) || !IS_ENABLED(CONFIG_FS_DAX_PMD))
        goto out;

    addr = __thp_get_unmapped_area(filp, len, off, flags, PMD_SIZE);
    if (addr)
        return addr;

 out:
    return current->mm->get_unmapped_area(filp, addr, len, pgoff, flags);
}
```

do_mmap申请地址空间时最终会调用到current->mm->get_unmmaped_area, bpf很难直接将这个地址打印出来，它到底是谁呢?

统信软件 UNIONTECH

2.7.3 将curtask的值打印出来

基于bpftrace工具写出如下跟踪代码

```
#!/usr/bin/bpftrace
BEGIN
{
    printf("begin trace nvme with kprobe, pid=%d\n", $1);
}
kprobe:thp_get_unmapped_area
{
    if (pid == $1) {
        printf("name=%s curtask=0x%llx\n", comm, curtask);
    }
}
END
{
    printf("finished\n");
}
```

上面的代码中curtask表示当前任务current

调用mmap函数的测试代码如下:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char** argv) {
    int fd=open("aa.mmap", O_RDWR);
    //mmap申请1G地址空间，触发左边thp_get_unmaped_area
函数的调用
    void *myaddr= mmap(0, 1024*1024*1024,
PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0);
    printf("fd=%d myaddr=0x%lx\n", fd, myaddr);
    getchar();//进程不退出，等待
    close(fd);
    return 0;
}
```

统信软件
UNIONTECH

2.7.4 运行bpf

假设上面的代码保存到了mmap.kp文件中

```
sudo ./mmap.kp 3899
Attaching 3 probes...
begin trace nvme with kprobe, pid=3899
name=mmap curtask=0xffff802f73dc5cc0
```

统信软件 UNIONTECH

2.7.5 基于gdb调试

sudo gdb ./vmlinux-163 /proc/kcore

将bpf中的curtask的地址直接拿过来使用

p ((struct task_struct*)0xffff802f73dc5cc0)->mm->get_unmapped_area

$3 = (unsigned long (*)(struct file *, unsigned long, unsigned long, unsigned long,

unsigned long)) 0xffff0000080a01d8 <arch_get_unmapped_area_topdown>

可以看到用gdb能够轻松得到get_unmmaped_area这个回调函数的地址。

总结：

bpf轻松得到当前进程的地址，gdb强大的调式能力可以得到该进程下面的所有的信息，

如：(gdb) p /x ((struct task_struct*)0xffffa02f96c9ea40)->mm->mmap_base

$3 = 0xfffff8000000

当前需要确保运行的内核和正在调试使用的vmlinux-163是一致的。

统信软件 UNIONTECH

2.8 总结

A)虽然内核可以使有GDB调试，但是搭建环境很麻烦，而且GDB调试时很容易让内核挂死。从我个人的实战来看，用GDB查看内核变量、内存值、反汇编代码等都比较方便，设置断点及跟踪调试还是不要期望太多。

B)GDB在调试内核上的不足BPF刚好可以弥补，BFP的强大的内核函数跟踪能力：包括调用堆栈、执行时间、函数调用参数地址、简单参数的值、进程task地址、进程名等等已经可以解决80%以上的内核调试需求了。

C)BPF的能够跟踪函数可以让我们快速确认此函数是否调用了、是谁调用的、调用时的参数是什么，如果数据结构类型的参数内容无法打印，可以打印出地址后借助GDB来查看。

D)对于内核调试，知道了进程的task地址也就基于上知道了这个进程的所有信息了，再借助GDB就可以轻松查看了。

E)GDB的查看也有局限性，它查看的是最终结果，如果想要看的值在函数调用过程中是不断变化的，那么最后用BPF的函数参数的打印能力来得知，但如果这个参数是内部数据结构需要显示就需要费一番功夫了，有可能自己写一个头文件可以实现，也有可能通过其它函数来得知。

F)在我眼里，BPF就是半个GDB， GDB的一半调试思想都可以用到BPF的调试上。

统信软件 UNIONTECH

3. ftrace

统信软件 UNIONTECH

3.1 引言

ftrace功能强大，android的整个系统的性能跟踪都是基于ftrace来做的，除了内核函数基于ftrace跟踪，framewrok层的大量关键函数都基于ftrace打点到内核进行跟踪,再配合基于浏览器运行的systrace工具非常容易分析framework层关键函数的耗时，如统计应用启动过程中解码图片多少次，在哪个线程完成，解码的图片文件名，每次用时多少等，非常方便。

目前uos上没有开发这么强大的性能统计工具，ftrace主要还只是内核支持(内核需要编译时支持，保证每个可ftrace的函数都加上了一条nop指令，同时内核的指令动态替换功能已经被编译进来等)。

因此本文基于ftrace的性能统计功能快速确认了copy_page优化点是否真的的变优了，这种统计技巧可以用到所有的能跟踪的函数的性能统计中，非常强大，好用。

但是ftrace仅仅统计内核也具有很大的局限性，如果采用默认的跟踪所有函数的机制，由于内核是最底层的函数，且有些函数运行过于频繁，导致性能下降很多，不方便分析每个函数的性能。不过基于这个机制可以分析内核函数的调用流程，查看堆栈等等。

统信软件
UNIONTECH

3.2 基于copy_page性能理解ftrace的使用

本节基于内核的copy_page来理解ftrace的使用，该函数基于汇编代码实现，比较底层，调用也比较频繁，但由于比较独立，因此比较方便当作例子来分析。

do_page_fault函数如果需要进行cow，最终会调用do_wp_page函数，do_wp_page进一步调用__cpu_copy_user_page函数。

本节代码将对三种不同的代码进行对比分析，分别是：

代码一：默认的copy_page函数

代码二：优化后的copy_page函数

代码三：采用另一种优化方案且改名为fast_copy_page

统信软件
UNIONTECH

3.3 默认的copy_page函数

```
ldp	x2, x3, [x1]
ldp	x4, x5, [x1, #16]
ldp	x6, x7, [x1, #32]
ldp	x8, x9, [x1, #48]
ldp	x10, x11, [x1, #64]
ldp	x12, x13, [x1, #80]
ldp	x14, x15, [x1, #96]
ldp	x16, x17, [x1, #112]

mov	x18, #(PAGE_SIZE - 128)
add	x1, x1, #128
1:
subs	x18, x18, #128

alternative_if ARM64_HAS_NO_HW_PREFETCH
	prfm	pldl1strm, [x1, #384]
alternative_else_nop_endif

stnp	x2, x3, [x0]
ldp	x2, x3, [x1]
stnp	x4, x5, [x0, #16]
ldp	x4, x5, [x1, #16]
stnp	x6, x7, [x0, #32]
ldp	x6, x7, [x1, #32]
stnp	x8, x9, [x0, #48]
ldp	x8, x9, [x1, #48]
stnp	x10, x11, [x0, #64]
ldp	x10, x11, [x1, #64]
stnp	x12, x13, [x0, #80]
ldp	x12, x13, [x1, #80]
stnp	x14, x15, [x0, #96]
ldp	x14, x15, [x1, #96]
stnp	x16, x17, [x0, #112]
ldp	x16, x17, [x1, #112]

add	x0, x0, #128
add	x1, x1, #128

b.gt	1b

stnp	x2, x3, [x0]
stnp	x4, x5, [x0, #16]
stnp	x6, x7, [x0, #32]
stnp	x8, x9, [x0, #48]
stnp	x10, x11, [x0, #64]
stnp	x12, x13, [x0, #80]
stnp	x14, x15, [x0, #96]
stnp	x16, x17, [x0, #112]

ret
```

1,302,416,732 instructions # 0.44 insn per cycle

统信软件
UNIONTECH

3.3.2 优化后的copy_page函数

```
ldp     x2, x3, [x1]
ldp     x4, x5, [x1, #16]
ldp     x6, x7, [x1, #32]
ldp     x8, x9, [x1, #48]
ldp     x10, x11, [x1, #64]
ldp     x12, x13, [x1, #80]
ldp     x14, x15, [x1, #96]
ldp     x16, x17, [x1, #112]

mov     x18, #(PAGE_SIZE - 128)
add     x1, x1, #128
1:
subs    x18, x18, #128

prfm    pldl1strm, [x1, #128]
prfm    pldl1strm, [x1, #192]

str     x2, [x0], #8
str     x3, [x0], #8
str     x4, [x0], #8
str     x5, [x0], #8
str     x6, [x0], #8
str     x7, [x0], #8
str     x8, [x0], #8
str     x9, [x0], #8
str     x10, [x0], #8
str     x11, [x0], #8
str     x12, [x0], #8
str     x13, [x0], #8
str     x14, [x0], #8
str     x15, [x0], #8
str     x16, [x0], #8
str     x17, [x0], #8

ldp     x2, x3, [x1]
ldp     x4, x5, [x1, #16]
ldp     x6, x7, [x1, #32]
ldp     x8, x9, [x1, #48]
ldp     x10, x11, [x1, #64]
ldp     x12, x13, [x1, #80]
ldp     x14, x15, [x1, #96]
ldp     x16, x17, [x1, #112]
add     x1, x1, #128
b.gt    1b

str     x2, [x0], #8
str     x3, [x0], #8
str     x4, [x0], #8
str     x5, [x0], #8
str     x6, [x0], #8
str     x7, [x0], #8
str     x8, [x0], #8
str     x9, [x0], #8
str     x10, [x0], #8
str     x11, [x0], #8
str     x12, [x0], #8
str     x13, [x0], #8
str     x14, [x0], #8
str     x15, [x0], #8
str     x16, [x0], #8
str     x17, [x0], #8
ret
```

1,302,416,732 instructions # 0.44 insn per cycle

统信软件 UNIONTECH

3.3.3 fast_copy_page

```
16 void __cpu_copy_user_page(void *kto, const void *kfrom, unsigned long vaddr)
17 {
18     struct page *page = virt_to_page(kto);

22     if(debug_fast_copy_page & 1){
23         kernel_neon_begin();
24         fast_copy_page(kto, kfrom);
25         kernel_neon_end();
26     }
27     else
28         copy_page(kto, kfrom);
29     flush_dcache_page(page);
30 }
```

红色代码为基于SIMD指令进行优化的代码，但是能够调用它之前需要先调用kernel_neon_begin进入neon模式

统信软件 UNIONTECH

3 .3.3 fast_copy_page

```
add   x8,x1,#0x20                    //x8=x1+0x20
add   x7,x1,#0x30                    //x7=x1+0x30
add   x6,x1,#0x40                    //x6=x1+0x40
add   x5,x1,#0x50                    //x5=x1+0x50
add   x4,x1,#0x60                    //x4=x1+0x60
add   x3,x1,#0x70                    //x3=x1+0x70
//先从源地址读到v0-v6矢量寄存器
ld1   {v6.2d}, [x2]                  //读16B数据到v6寄存器. 0x10
ld1   {v5.2d}, [x8]                  //读16B数据到v5寄存器  0x20
ld1   {v4.2d}, [x7]                  //读16B数据到v4寄存器  0x30
ld1   {v3.2d}, [x6]                  //读16B数据到v3寄存器  0x40
ld1   {v2.2d}, [x5]                  //读16B数据到v2寄存器  0x50
ld1   {v1.2d}, [x4]                  //读16B数据到v1寄存器  0x60
ld1   {v0.2d}, [x3]                  //读16B数据到v0寄存器  0x70
//通过v0-v7矢量寄存器写到目的地址，x0指向的是目的地址首地址
//x0指向的都是目的地址，v7.2d是前16B的数据，写完后x0=x0+16
st1   {v7.2d}, [x0], #16
st1   {v6.2d}, [x0]                  //写v6.2d到16偏移位置
add   x0,x19, #0x20                  //x0=x19+0x20, x0代表目的地
st1   {v5.2d}, [x0]                  //可以调用st1写目的地
add   x0,x19, #0x30
st1   {v4.2d}, [x0]
add   x0,x19, #0x40
st1   {v3.2d}, [x0]
add   x0,x19, #0x50
st1   {v2.2d}, [x0]
add   x2,x19, #0x60
add   x0,x19, #0x70
st1   {v1.2d}, [x2]
add   x19,x19, #0x80
st1   {v0.2d}, [x0]
```

统信软件
UNIONTECH

3 .4 基于ftrace跟踪

a)设置trace跟踪器
echo function_graph > /sys/kernel/debug/tracing/current_tracer

查看可用的跟踪器：
cat /sys/kernel/debug/tracing/available_tracers
hwlat blk mmiotrace function_graph wakeup_dl wakeup_rt wakeup function nop

b)设置最大跟踪深度，可以避免内部函数嵌套太多，设置为 1 时只跟踪函数本身
echo 2 > /sys/kernel/debug/tracing/max_graph_depth

c)设置需要跟踪的函数，避免所有函数都被跟踪
echo do_wp_page > /sys/kernel/debug/tracing/set_graph_function

d)开始跑测试用例
./pgms/spawn 30

e)获取跟踪信息
cat traces > /sys/kernel/debug/tracing/ftrace_dowppage.log

统信软件
UNIONTECH

3 .4.2 默认copy_page性能统计

```
3)                |          __cpu_copy_user_page() {
3)    0.562 us    |            flush_dcache_page();
3) + 10.646 us    |          }

3)                |          __cpu_copy_user_page() {
3)    0.500 us    |            flush_dcache_page();
3)    3.500 us    |          }

3)                |          __cpu_copy_user_page() {
3)    0.520 us    |            flush_dcache_page();
3)    3.500 us    |          }

3)                |          __cpu_copy_user_page() {
3)    0.541 us    |            flush_dcache_page();
3)    2.709 us    |          }

3)                |          __cpu_copy_user_page() {
3)    0.500 us    |            flush_dcache_page();
3)    3.479 us    |          }
```

copy_page是汇编代码，不能被直接跟踪，
因此ftrace中没有将copy_page例举出来。

但是从__cpu_copy_user_page的函数实现来看，
仅仅调用了两个函数:copy_page和flush_dcache_page
函数，因此用总函数的时间减去flush_dcache_page函数
的时间就可以了。

从上面的统计时间来看:
copy_page的时间在2.2us-3us之间
第一次的10us时间是 一个巨大抖动
但这也说明了这种抖动是存在的
大部分时间集中在3us左右

1,302,416,732 instructions # 0.44 insn per cycle

统信软件
UNIONTECH

3 .4.3 优化后copy_page性能统计

第一次

```
 0)                |          __cpu_copy_user_page() {
 0)    0.521 us    |            flush_dcache_page();
 0)    2.855 us    |          }
copy_page=2.32s
```

从统计来看，
运行时间在1.8us-2.3us之间，
优化明显。

第二次：

```
 0)                |          __cpu_copy_user_page() {
 0)    0.521 us    |            flush_dcache_page();
 0)    2.417 us    |          }
```

第三次：

```
 0)                |          __cpu_copy_user_page() {
 0)    0.521 us    |            flush_dcache_page();
 0)    2.334 us    |          }
```

1,881,373,684 instructions # 1.46 insn per cycle

统信软件 UNIONTECH

3 .4.4 fast_copy_page性能统计

第一次：

```
 0)              |     __cpu_copy_user_page() {
 0)              |       kernel_neon_begin() {
 0)   0.541 us   |         __local_bh_disable_ip();
 0)   0.563 us   |         fpsimd_save();
 0)   0.521 us   |         fpsimd_flush_cpu_state();
 0)   0.771 us   |         __local_bh_enable_ip();
 0)   5.105 us   |       }
 0)   0.521 us   |       kernel_neon_end();
 0)   0.520 us   |       flush_dcache_page();
 0)   8.833 us   |     }
```

第二次：

```
 0)              |     __cpu_copy_user_page() {
 0)              |       kernel_neon_begin() {
 0)   0.541 us   |         __local_bh_disable_ip();
 0)   0.563 us   |         fpsimd_save();
 0)   0.521 us   |         fpsimd_flush_cpu_state();
 0)   0.750 us   |         __local_bh_enable_ip();
 0)   5.063 us   |       }
 0)   0.542 us   |       kernel_neon_end();
 0)   0.520 us   |       flush_dcache_page();
 0)   8.916 us   |     }
```

以第一次为例，
__cpu_copy_user_page函数的总时间减去能统计出来的时间就是fast_copy_page函数的时间了。
8.833-5.105-0.521-0.520=2.69us

第二次：8.916-5.063-0.542-0.520=2.79us

第三次：9.25-5.104-0.521-0.938=2.69us.

从统计时间来看，它没有优化后的copy_page快
即没有代码二快
所以代码二对应的copy_page的性能是最好的

统信软件
UNIONTECH

3 .5 对copy_page本身的总结

1)代码一中的copy_pag基于ldp/stnp来实现的，它们每次都读写16B.
但是由于ldp和stnp都是操作读写同一个寄存器，insn指标偏低，才0.44

2)代码二将stnp修改成str, 虽然每次只写8B, 由于str是同时写不同的寄存器，并行性很好，
因此性能反而比代码一中的stnp代码更优
基于perf stat 统计发现，此insn为1.46

3)代码三优化的原理就是基于neon的SIMD指令来写数据，这儿优化使用的核心指令就是
ld1和st1。ld1一次读16B, st1一次写16B, 相对于ldnp和stnp指令最大的区别就是后者
一个时钟周期之写8B, 虽然ldnp 可以带两个寄存器一次写16B, 但不是一个时钟周期完成
的。

4)通过将fast_copy_page汇编代码放到用户态运行，性能比基于ldnp和stnp的性能好，
基于ldnp和stnp的copy_page性能抖动比较明显，假设ld1/st1的性能在2us, 那么
ldnp/stnp的性能就在2-4us了。

但由于使用ld1, st1指令之前需要先调用kernel_neon_begin，结束后调用
kernel_neon_end, 这两个函数本身性能偏低，导致整个性能下降，通过前面的ftrace测
试发现内核模式下该fast_copy_page的性能是最慢的

统信软件
UNIONTECH

3 .6 对ftrace的总结

1)ftrace具有强大的性能统计功能，可以好好利用。ftrace可以跟踪内核所有的除汇编代码以为的函数（部分统计函数及上下问切换函数除外），因此这些函数的执行性能可以全部统计出来。

2)当完成某个底层函数的优化后，应该先基于ftrace确认一下性能是否真的变优了，为了方便调试，可以先在用户态写出来调试，待稳定后再移到内核态。网上很多补丁声称优化很多，但是真正测试发现有时不但没有提升还会恶化。因此基于ftrace快速确认是一个比较好的习惯

3)对于汇编级别的优化代码，虽然基于ftrace无法直接统计，但是可以统计调用它的函数，如上面的统计copy_page汇编代码的性能时，基于do_wp_page来统计，通过几个加减运算就可以确定copy_page函数的性能了。

4)bfp也具有性能跟踪的功能，但是和ftrace不同的是，它无法跟踪到它里面调用的函数，需要单独跟踪，如对于fast_copy_page函数中看到的先调用copy_page, 再调用flush_dchace_page这个函数，copy_page是汇编代码，无法跟踪，用bpf只能独立跟踪__cpu_copy_user_page和flush_dcache_page, 但是此时很难一一对应上，而用ftrace就不用担心此问题了。

统信软件
UNIONTECH

4. strace

统信软件
UNIONTECH

4.1 引言

所有的系统调用都可以跟踪吗?可以的,它就是strace， strace可以跟踪所有的系统调用。

所谓系统调用就是c库本身已经无法完成，需要请求内核去完成，这个请求内核去完成的调用就是系统调用，如打开一个文件，因为最终的文件创建、读写等都需要内核的文件系统来完成；还有如mmap申请一个内存，内存也是由内核的内存管理大系统完成的。

从我个人经验来看，strace使用最常见场景有：

跟踪系统级调用流程：如是否创建了子进程、打开了多少次文件、mmap调用多少次、设置过什么样的权限等等 。

基于sandbox环境做安全调试：sandbox主要基于文件系统做安全管控，strace也可以跟踪文件的所有操作，如打开某个文件不存在、没有权限都可以通过strace快速得知。

性能分析：strace可以打印每次系统调用的时间，基于时间能发现耗时的系统调用、基于次数可以发现频繁的系统调用。

strace很强大，很方便使用，strace是基于什么原理实现的呢?

本节先介绍strace的实现原理，基于gdb快速理解其实现原理，除了理解其原理本身，如何用gdb跟踪的思路也值得学习。

最后通过两个例子学习如何通过strace灵活分析问题。

统信软件
UNIONTECH

4.2 strace实现原理

先下载一份strace源代码，然后编译出调试版本，最后基于gdb调试及查看内核代码是最快的了解方式。

1)uos系统下载源代码

apt source strace

2)生成makefile

./configure --enable-mpers=no

mpers给系统调用加上前缀，加上编译不通过，直接disable mpers

3)make

此时生成一个-g -O2版本的strace, 很方便用gdb调试.

strace默认生成strace文件夹下面

统信软件
UNIONTECH

4.2.2 strace实现原理

gdb调试之前先大致看了一下strace下面的main函数、init函数，init函数设置了信号处理，也调用ptrace函数，因此初步估计strace基于ptrace来完成系统调用的跟踪的，但是实现逻辑还不清楚别是ftrace的实现需要内核做什么刚好可以基于strace来理解一下了。

strace依赖大量文件，如open.c这样的文件，初步估计它是为了跟踪open系统调用。基它文件当然就是为了完成其它系统调用了。

open.c中看到了如下代码：

```
SYS_FUNC(openat)
{
    print_dirfd(tcp, tcp->u_arg[0]);
    return decode_open(tcp, 1);
}
```

SYS_FUNC 展开后就是sys_openat

```
nm strace|grep sys_openat
000000000041e2f0 T sys_openat
```

因此写一个简单例子，调用open, 触发openat的系统调用，然后基于gdb启动跟踪sys_open函数，看看是如何被调用的。

统信软件
UNIONTECH

4.2.3 调用open的简单例子

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char** argv) {
    int fd=open("aa.txt", O_CREAT|O_RDWR);//触发openat系统调用
    printf("fd=%d\n",fd); //触发write系统调用
    close(fd);//触发close系统调用
    return 0;
}
```

统信软件 UNIONTECH

4.2.4 gdb调试sys_openat

gdb /home/zhoupeng/linux/linux/code/strace/strace-4.26/strace

启动成功后，设置traceopen

set args ./traceopen

对sys_open设置断点，因为想看看sys_open被调用的逻辑

b sys_openat

运行结果得到如下堆栈

```
Breakpoint 3, sys_openat (tcp=0x4d8570) at open.c:123
123         print_dirfd(tcp, tcp->u_arg[0]);
(gdb) bt
#0  sys_openat (tcp=0x4d8570) at open.c:123
#1  0x00000000004319a1 in syscall_entering_trace (tcp=tcp@entry=0x4d8570, sig=sig@entry=0x7fffffffc934) at syscall.c:643
#2  0x000000000042e8db in trace_syscall (tcp=0x4d8570, sig=0x7fffffffc934) at strace.c:2424
#3  0x0000000000430b31 in dispatch_event (wd=<optimized out>) at strace.c:2463
#4  0x00000000004028e7 in main (argc=<optimized out>, argv=<optimized out>) at strace.c:2289
(gdb)
```

统信软件 UNIONTECH

4.2.5 gdb调试sys_openat

```
(gdb) bt
#0  sys_openat (tcp=0x4d8570) at open.c:123
#1  0x00000000004319a1 in syscall_entering_trace (tcp=tcp@entry=0x4d8570, sig=sig@entry=0x7fffffffc934) at syscall.c:643
#2  0x000000000042e8db in trace_syscall (tcp=0x4d8570, sig=0x7fffffffc934) at strace.c:2424
#3  0x0000000000430b31 in dispatch_event (wd=<optimized out>) at strace.c:2463
#4  0x00000000004028e7 in main (argc=<optimized out>, argv=<optimized out>) at strace.c:2289
(gdb) up
#1  0x00000000004319a1 in syscall_entering_trace (tcp=tcp@entry=0x4d8570, sig=sig@entry=0x7fffffffc934) at syscall.c:643
643             int res = raw(tcp) ? printargs(tcp) : tcp->s_ent->sys_func(tcp);
(gdb) up
#2  0x000000000042e8db in trace_syscall (tcp=0x4d8570, sig=0x7fffffffc934) at strace.c:2424
2424                            res = syscall_entering_trace(tcp, sig);
(gdb) up
#3  0x0000000000430b31 in dispatch_event (wd=<optimized out>) at strace.c:2463
2463                    if (trace_syscall(current_tcp, &restart_sig) < 0) {
(gdb) l 2460,2465
2460                    break;
2461
2462            case TE_SYSCALL_STOP:
2463                    if (trace_syscall(current_tcp, &restart_sig) < 0) {
2464                            /*
2465                             * ptrace() failed in trace_syscall().
(gdb)
```

1)up命令，回退到上一层调用

2) 继续回退到上一层调用

3) 继续回退到了调用trace_syscall的地方

4)显示2460~ 2465这几行的代码

5)调用trace_syscall是因为TE_SYSCALL_STOP这个命令
所以继续跟踪是谁设置了TE_SYSCALL_STOP

4.2.6 查看TE_SYSCALL_STOP的设置与处理

grep搜索发现next_event完成设置，也就是说搜索到一个事件后就调用next_event函数。

看看next_event函数的关键流程

```
static const struct tcb_wait_data * next_event(void) {
    switch (event) {
        case 0:
            ...
            if (sig == syscall_trap_sig) {
                wd->te = TE_SYSCALL_STOP;
            }
        case PTRACE_EVENT_STOP:
                ...
        case PTRACE_EVENT_EXEC:
                ...
    }
}
const unsigned int syscall_trap_sig = SIGTRAP | 0x80;
```

next_event返回值给dispatch_event使用，如下：

```
dispatch_event(const struct tcb_wait_data *wd) {
    enum trace_event te = wd ? wd->te : TE_BREAK;
    switch (te) {
      case TE_BREAK:       return false;
      case TE_NEXT:        return true;
      case TE_RESTART:     break;
      case TE_SYSCALL_STOP:
        if (trace_syscall(current_tcp, &restart_sig) < 0) {
            return true;
      }...
}
```

可以猜测试next_event调用是因为收到了SIGTRAP信号。

此信号表示的事件id=0, strace转换成了TE_SYSCALL_STOP，从名字的定义来看就是系统调用要暂停一下，就像gdb遇到了一个断点一样，不一样的是gdb让我们去调试，这儿直接调用对应的处理函数了。

这儿调用的处理函数就是trace_syscall。

统信软件
UNIONTECH

4.2.7 查找ptrace的调用

从前面的分析可以知道，strace设置的sys_openat函数能够被调用是因为收到了SIGTRAP信号，收到此信号需要向内核发出ptrace请求，因此跟踪ptrace表求的调用，看看strace是在哪儿发出来的。

gdb继续调试

b ptrace

r 重新运行，结果看到了向内核发出PTRACE_SYSCALL请求的系统调用

ptrace (request=request@entry=PTRACE_SYSCALL) at ../sysdeps/unix/sysv/linux/ptrace.c:30

30 ../sysdeps/unix/sysv/linux/ptrace.c: 没有那个文件或目录.

(gdb) bt

#0 ptrace (request=request@entry=PTRACE_SYSCALL) at ../sysdeps/unix/sysv/linux/ptrace.c:30

#1 0x0000000000430206 in ptrace_restart (op=24, sig=0, tcp=<optimized out>) at strace.c:356

#2 0x0000000000430b60 in dispatch_event (wd=<optimized out>) at strace.c:2579

#3 0x00000000004028e7 in main (argc=<optimized out>, argv=<optimized out>) at strace.c:2289

dispatch_event --> ptrace_restart

ptrace_restart中的代码如下：

ptrace(op, tcp->pid, 0L, (unsigned long) sig);

tcp->pid是要被跟踪的进程

结论：

strace进程已经向内核发出来PTRACE_SYSCALL请求，现在看看内核是如何处理的。

统信软件
UNIONTECH

4.2.8 内核对PTRACE_SYSCALL处理

从 SYSCALL_DEFINE4(ptrace...)出发，其调用流程如下：

sys_ptrace --> arch_ptrace --> ptrace_request --> ptrace_resume

ptrace_resume中设置了TIF_SYSCALL_TRACE跟踪标志：

set_tsk_thread_flag(child, TIF_SYSCALL_TRACE);

再看使用TIF_SYSCALL_TRACE的地方

以arm64为例：

./arch/arm64/kernel/syscall.c

el0_svc_common --> syscall_trace_enter

syscall_trace_enter中有如下调用

```
if (test_thread_flag(TIF_SYSCALL_TRACE))
    tracehook_report_syscall(regs, PTRACE_SYSCALL_ENTER);
```

结论：

el0_svc_common是系统调用的入口函数，调用之前先检查TIF_SYSCALL_TRACE标志，如果有此标志，则调用tracehook_report_syscal报告此系统调用

统信软件 UNIONTECH

4.2.9 确认内核的syscall上报

从 tracehook_report_syscall出发，其调用流程如下：

tracehook_report_syscall --> tracehook_report_syscall_entry --> ptrace_report_syscall

ptrace_report_syscall中有如下调用：

ptrace_notify(SIGTRAP | ((ptrace & PT_TRACESYSGOOD) ? 0x80 : 0));

结论：

当系统调用被跟踪时，果然向监听者发出来SIGTRAP信号，且退出码为SIGTRAP_0x80

整个内核的实现比这复杂、全面，如需要先调用ptrace进入跟踪状态，SYSCALL只是其中一个请求，还有LISTEN请求等；除了进入此函数能发出SIGTRAP信号，退出系统调用时也会发出，这个是strace能够统计此函数调用性能原因。

这些细节都可以从tracehook_report_syscall出发来了解。

strace的这个跟踪能力和gdb的实现原理一样，只是gdb比这做得更多，都依赖内核的ptrace能力完成。所以全面理解后基于此原理实现一个gdb也不是没有可能。

统信软件
UNIONTECH

4.3 基于strace理解系统调用

strace就是对用态调用函数进行跟踪，所以系统调用函数的名字，参数，返回值等信息信息全部都能得到。

写一个调用open函数的测试代码：

```
int main(int argc, char** argv)
{
        int aa=O_CREAT;
        int bb = O_RDWR;
        int fd = open("/home/uos/456.hmmer/aa.txt", O_CREAT|O_RDWR);
        if (fd > 0)
            close (fd);

        fd = open("aa.txt", O_CREAT|O_RDWR);
        if (fd > 0) close(fd);
        return 0;
}
```

统信软件
UNIONTECH

4.3.2 跟踪运行

gcc -g testopen.c -o testopen

strace ./testopen

strace中看到如下三条系统调用：

openat(AT_FDCWD, "/home/uos/456.hmmer/aa.txt", O_RDWR|O_CREAT, 01000) = 3

close(3)   = 0

openat(AT_FDCWD, "aa.txt", O_RDWR|O_CREAT, 01000) = -1 EACCES (权限不够)

1)open("/home/uos/456.hmmer/aa.txt", O_CREAT|O_RDWR)对应的系统调用是

openat(AT_FDCWD, "/home/uos/456.hmmer/aa.txt", O_RDWR|O_CREAT, 01000) = 3

此时打开文件时返回的fd为 3

2)由于打开文件成功，所以close关闭的fd等于3

3)open("aa.txt", O_CREAT|O_RDWR)对应的系统调用是

openat(AT_FDCWD, "aa.txt", O_RDWR|O_CREAT, 01000) = -1 EACCES (权限不够)

返回错误是因为aa.txt没有返回权限

-r-S--S--T 1 uos uos 7 5月 13 17:03 aa.txt

统信软件
UNIONTECH

4.3.3 理解系统调用

第一个疑问:

代码中为open，为什么系统调用时看到的是openat，close调用看到的却还是close这个函数原型。

首先基于gdb确认调用的c库函数

b open

Breakpoint 2, __libc_open64 (

file=0x400798 "/home/uos/456.hmmer/aa.txt", oflag=66) at ../sysdeps/unix/sysv/linux/open64.c:37

(gdb) p open

$1 = {int (const char *, int, ...)} 0xfffff7ec1f78 <__libc_open64>

堆栈看到的函数是__libc_open64，自己写的代码调用的是open函数，这个是什么原因呢？这个是c库的弱符号函数导致的，真正的实现是__libc_open64, 下面的四个函数只是__libc_open64的弱符号别名

00000000000c3390 t __libc_open64

00000000000c3390 W __open

00000000000c3390 W open

00000000000c3390 W __open64

00000000000c3390 W open64

统信软件
UNIONTECH

4.3.4 理解系统调用

理解__libc_open函数中最终是如何进行系统调用的。

__libc_open函数在./unix/sysv/linux/open64.c

最终的系统调用如下：

```
return SYSCALL_CANCEL (openat, AT_FDCWD, file, oflag | EXTRA_OPEN_FLAGS,
        mode);
```

系统调用对应的函数是openat, 这个是strace中看到openat系统调用的根因

close函数最终的系统调用如下：

```
 return SYSCALL_CANCEL (close, fd);
```

由于系统调用的名字就是close, 所以strace中看到的系统调用就是close了

统信软件 UNIONTECH

4.3.5 理解系统调用

基于预编译机制看一下此宏展开时的情况：

需要理解的是对于openat系统调用,系统调用号56通过寄存器x8传入

```
    return ({ long int sc_ret; if (__builtin_expect ((__libc_multiple_threads == 0), 1)) sc_ret = ({ unsigned long _sys_res
ult = ({ long _sys_result; { long _x3tmp = (long) (mode); long _x2tmp = (long) (oflag | 0); long _x1tmp = (long) (file); lo
ng _x0tmp = (long) (-100); register long _x0 asm ("x0"); _x0 = _x0tmp; register long _x1 asm ("x1") = _x1tmp; register long
 _x2 asm ("x2") = _x2tmp; register long _x3 asm ("x3") = _x3tmp; register long _x8 asm ("x8") = ((
# 66 "../sysdeps/unix/sysv/linux/libopen.c" 3 4
          56
# 66 "../sysdeps/unix/sysv/linux/libopen.c"
          )); asm volatile ("svc    0   // syscall " "SYS_ify(openat)" : "=r" (_x0) : "r"(_x8) , "r" (_x0), "r" (_x1), "r"
(_x2), "r" (_x3) : "memory"); _sys_result = _x0; } _sys_result; }); if (__builtin_expect (((unsigned long) (_sys_result) >=
 (unsigned long) -4095), 0)) { (__libc_errno = ((-(_sys_result)))); _sys_result = (unsigned long) -1; } (long) _sys_result;
 }); else { int sc_cancel_oldtype = __libc_enable_asynccancel (); sc_ret = ({ unsigned long _sys_result = ({ long _sys_resu
lt; { long _x3tmp = (long) (mode); long _x2tmp = (long) (oflag | 0); long _x1tmp = (long) (file); long _x0tmp = (long) (-10
0); register long _x0 asm ("x0"); _x0 = _x0tmp; register long _x1 asm ("x1") = _x1tmp; register long _x2 asm ("x2") = _x2tm
p; register long _x3 asm ("x3") = _x3tmp; register long _x8 asm ("x8") = ((
# 66 "../sysdeps/unix/sysv/linux/libopen.c" 3 4
          56
# 66 "../sysdeps/unix/sysv/linux/libopen.c"
          )); asm volatile ("svc    0   // syscall " "SYS_ify(openat)" : "=r" (_x0) : "r"(_x8) , "r" (_x0), "r" (_x1), "r"
(_x2), "r" (_x3) : "memory"); _sys_result = _x0; } _sys_result; }); if (__builtin_expect (((unsigned long) (_sys_result) >=
 (unsigned long) -4095), 0)) { (__libc_errno = ((-(_sys_result)))); _sys_result = (unsigned long) -1; } (long) _sys_result;
 }); __libc_disable_asynccancel (sc_cancel_oldtype); } sc_ret; })
```

1)对于openat，56这个系统调用号赋值给了x8寄存器

2）使用svc指令进入系统调用

统信软件 UNIONTECH

4.3.6 理解系统调用

再看看此时的汇编代码，系统调用号确实写到了x8寄存器。最后调用svc 0进入系统调用

理解open系统调用的实现. open函数的系统调用号为56

```
0x0000fffff7ec1fcc <+84>:    mov    x1, x20
0x0000fffff7ec1fd0 <+88>:    mov    x0, #0xffffffffffffff9c        // #-100
0x0000fffff7ec1fd4 <+92>:    mov    x8, #0x38                      // #56
0x0000fffff7ec1fd8 <+96>:    svc    #0x0          openat的系统调用号为56
```

理解close系统调用的实现, close函数的系统调用号为57

```
0x0000fffff7ec2a34 <+28>:    cbnz   w0, 0xfffff7ec2a60 <__GI___close+72>
0x0000fffff7ec2a38 <+32>:    mov    x0, x19
0x0000fffff7ec2a3c <+36>:    mov    x8, #0x39                      // #57
0x0000fffff7ec2a40 <+40>:    svc    #0x0   系统调用号为57
```

统信软件
UNIONTECH

4.3.7 理解系统调用

系统调用是如何实现的呢？如何最终调用到内核的open函数的呢？内核的函数又是谁呢?

一般内核的系统函数都会在c库接口函数前面加上sys关键字，对于arm 64系统还会进一步加上arm64前缀。

可以快速查找/proc/kallsyms来确认

```
sudo cat /proc/kallsyms |grep sys_open
ffff0000082925e0 T __arm64_sys_open
```

arm系统上最终调用的是__arm64_sys_open函数

基于bpf 跟踪此函数调用堆栈：

```
sudo trace-bpfcc -tK  __arm64_sys_openat -p 4681
TIME     PID     TID     COMM          FUNC
2.320917 4681    4681    testopen      __arm64_sys_openat
        __arm64_sys_openat+0x0 [kernel]
        el0_svc_handler+0x9c [kernel]
        el0_svc+0x8 [kernel]
```

统信软件
UNIONTECH

4.3.8 理解系统调用

理解el0_svc_handler函数的实现

```
163 asmlinkage void el0_svc_handler(struct pt_regs *regs)
164 {
165     sve_user_discard();
166 #ifdef CONFIG_EXAGEAR_BT
167     if (regs->regs[8] & 0x80000000) {
168         current_thread_info()->exagear_syscall = 1;
169         el0_svc_common(regs, regs->regs[7], __NR_compat_syscalls,
170                        compat_sys_call_table);
171     } else
172 #endif
173     el0_svc_common(regs, regs->regs[8], __NR_syscalls, sys_call_table);
174 }
```


regs->regs[8]刚好就是前面x8宏存器，系统调用号保存到了此寄存器中，现在需要读出来使用。

统信软件
UNIONTECH

4.3.9 理解系统调用

看sys_call_table的赋值

```
const syscall_fn_t sys_call_table[__NR_syscalls] = {
    [0 ... __NR_syscalls - 1] = __arm64_sys_ni_syscall,
#include <asm/unistd.h>
};
"arch/arm64/kernel/sys.c" 行 58 / 61 --95%-- 列 1
```

```
const syscall_fn_t sys_call_table[294] = {
 [0 ... 294 - 1] = __arm64_sys_ni_syscall,//这儿先对整个数组设置默认值
[0] = __arm64_sys_io_setup,
[1] = __arm64_sys_io_destroy,
...
[56] = __arm64_sys_openat,//上面的asm/unistd.h头文件中的定义完成了这些函数的定义
[57] = __arm64_sys_close,
...
}
```

统信软件
UNIONTECH

4.3.10 理解系统调用总结

总结:

a)系统调用的本质就是用户态通过寄存器传入系统调用号，然后调用特权指令进入内核态运行。内核态基于系统调用号从系统函数表中查找到对应的函数，然后运行它。

b)对于内核，系统调用号在./include/uapi/asm-generic/unistd.h文件中定定义。

如： #define __NR_openat 56

__SC_COMP(__NR_openat, sys_openat, compat_sys_openat)

arm64版本上最后生成的是：[56] = __arm64_sys_openat

c)对于c库，系统调用号在include/asm-generic/unistd.h文件中定义

如:

#define __NR_openat 56

__SC_COMP(__NR_openat, sys_openat, compat_sys_openat)

和内核中的定义一样

d)strace本身的使用比较简单，本质都是跟踪系统调用，因此这个例子重点讲解了如何根据系统调用找到对应的c库函数，系统调用是如保实现的。

统信软件
UNIONTECH

4.4 strace解决打开文件失败问题

还是用前面的traceopen.c中的例子，打开文件为aa.txt

将aa.txt的owner改成root

sudo chown root aa.txt

strace ./traceopen

```
mprotect(0xfffcbf3e0000, 65536, PROT_READ) = 0
munmap(0xfffcbf380000, 98407)           = 0
openat(AT_FDCWD, "aa.txt", O_RDWR|O_CREAT, 0112530) = -1 EACCES (权限不够)
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
brk(NULL)                               = 0x20ee0000
brk(0x20f10000)                         = 0x20f10000
write(1, "fd=-1\n", 6fd=-1
)                   = 6
```

提示权限不够

统信软件 UNIONTECH

4.5 总结

a) 本节主要介绍了strace的实现原理，没有介绍strace如何使有，因为如何使用的资料很多，也可以直接strace -h查看

b)strace本质上依赖内核提供的ptrace功能实现，ptrace实现的本质就是hook跟踪。 对SYSCALL这个请求而言，内核对所有系统调用都通知出来，且以SIGTRAP信号通知到监听者。

c)了解了strace的实现原理，gdb的实现原理也就好理解了，同样基于ptrace实现。只是strace收到SIGTRAP信号然后调用内部的实现函数运行，gdb对我们的断点设置ptrace, 当收到SIGTRAP信号后，停止来下，让我们去调试。

d)由于strace跟踪的是系统调用，所以本节也详细介绍了系统调用的实现原理，其本质就是查表调用函数。实现逻辑为：每个系统调用有一个调用号，调用号及其相应的参数都基于汇编代码通过寄存器传入, svc特权指令进入内核，内核调用el0_svc来处理，最后基于sys_call_table[系统调用号]这种方式来处理，比较像中断处理流程。

统信软件
UNIONTECH

5. vmtouchX

统信软件
UNIONTECH

5.1 引言

文件缓存的精确统计是io相关类问题性能优化不可缺少的一环，因为影响io性能的主要点是磁盘和内存，内存的读密写速度通常比磁盘快，机械硬盘环境情况下，这个性能差可以达到10倍以上。所以当io读写基于文件系统缓存时，此时的优化基本就是如何充分利用缓存的优化了。

可执行进程、动态库、资源文件也是文件，这些文件提前缓存在内存也有助于加速进程的启动的，特别是在基于低速flash的设备上。

查看文件系统缓存的工具常用的有free, fincore, vmtouch, free只能统计整个系统的缓存情况，fincore和vmtouch能统计单个文件的缓存。

在多node以及高负载级存场景下，即使是能够统计单个文件缓存的fincore, vmtouch也很难精准分析缓存相关问题。如多node 服务器上进行10G大文件copy，虽然都缓存了，为什么存在很明显的性能抖动；iozone两倍物理内存，按照缓存先进先出的回收机制，为什么也存在20%以下的性能抖动，还有优化的空间吗?

本文从这两个问题出发，分析了背后的原因，并介绍了如何完善vmtcouch帮助我们精准分析缓存相关的问题。

统信软件 UNIONTECH

5.2 效果展示

1)首先创建一个1G的文件，且保证在文件系统中缓存

dd if=/dev/zero of=1G.txt bs=1G count=1

2)使用vmtouch统计此文件的缓存情况

./vmtouch/vmtouch 1G.txt

```
           Files: 1
     Directories: 0
  Resident Pages: 262144/262144  1G/1G  100%
         Elapsed: 0.013526 seconds
```

1)统计了一个缓存文件

2)缓存100%

3) 整个文件分段统计，每个占100M，全部缓存。当一个文件只有部分缓存时，这种分段统计的功能非常有助于分析问题

		pinzone	pinfile	node0	node1	node2	node3
1	:	100.00%	10.00%	100.00%	0.00%	0.00%	0.00%
2	:	100.00%	10.00%	100.00%	0.00%	0.00%	0.00%
3	:	100.00%	10.00%	100.00%	0.00%	0.00%	0.00%
4	:	100.00%	10.00%	100.00%	0.00%	0.00%	0.00%
5	:	100.00%	10.00%	100.00%	0.00%	0.00%	0.00%
6	:	100.00%	10.00%	100.00%	0.00%	0.00%	0.00%
7	:	100.00%	10.00%	100.00%	0.00%	0.00%	0.00%
8	:	100.00%	10.00%	100.00%	0.00%	0.00%	0.00%
9	:	100.00%	10.00%	100.00%	0.00%	0.00%	0.00%
10	:	100.00%	10.00%	100.00%	0.00%	0.00%	0.00%

5.2.2 效果展示

1)创建一个1G的文件，且基于oflag不经过缓存方式写文件

dd if=/dev/zero of=1G.txt bs=1G count=1 oflag=direct

2)使用vmtouch统计此文件的缓存情况

./vmtouch/vmtouch 1G.txt

```
uos@uos-PC:~/zhoupeng$ ./vmtouch/vmtouch ./1G.txt
           Files: 1
     Directories: 0
  Resident Pages: 0/16384  0/1G  0%      1)缓存为0
         Elapsed: 0.000122 seconds


                          2) 没有一页级存
 :
      pinzone      pinfile   node0    node1    node2    node3
1   :   0.00%    0.00%    0.00%    0.00%    0.00%    0.00%
2   :   0.00%    0.00%    0.00%    0.00%    0.00%    0.00%
3   :   0.00%    0.00%    0.00%    0.00%    0.00%    0.00%
4   :   0.00%    0.00%    0.00%    0.00%    0.00%    0.00%
5   :   0.00%    0.00%    0.00%    0.00%    0.00%    0.00%
6   :   0.00%    0.00%    0.00%    0.00%    0.00%    0.00%
7   :   0.00%    0.00%    0.00%    0.00%    0.00%    0.00%
8   :   0.00%    0.00%    0.00%    0.00%    0.00%    0.00%
9   :   0.00%    0.00%    0.00%    0.00%    0.00%    0.00%
10  :   0.00%    0.00%    0.00%    0.00%    0.00%    0.00%
```

5.3 缓存工具分析iozone

1)以iozone为例

./iozone -+n -s 64g -r 16m -w -i 0

iozone.tmp为已经跑完写2倍物理内存的数据后的文件，vmtouch缓存统计结果如下：

```
root@uos-PC:/home/uos/cyc/vmtouch# ./vmtouch /home/uos/cyc/iozone/src/iozone.tmp
           Files: 1
     Directories: 0
  Resident Pages: 7563699/16777216  28G/64G  45.1%
         Elapsed: 1.4744 seconds

   :
        pinzone     pinfile    node0     node1     node2     node3
1    :   0.00%      0.00%      0.00%     0.00%     0.00%     0.00%
2    :   0.00%      0.00%      0.00%     0.00%     0.00%     0.00%
3    :   0.00%      0.00%      0.00%     0.00%     0.00%     0.00%
4    :   0.00%      0.00%      0.00%     0.00%     0.00%     0.00%
5    :   4.27%      0.43%      0.00%     0.00%     4.27%     0.00%
6    :   46.56%     4.66%      31.17%    0.00%     15.39%    0.00%
7    :   100.00%    10.00%     46.97%    0.00%     53.03%    0.00%
8    :   100.00%    10.00%     21.48%    0.00%     78.52%    0.00%
9    :   100.00%    10.00%     87.74%    0.00%     12.26%    0.00%
10   :   100.00%    10.00%     31.83%    0.00%     68.17%    0.00%
root@uos-PC:/home/uos/cyc/vmtouch#
```

1)整个文件64G,但只有28G内容在缓存里面

2）文件的前半部分基本上没有缓存，全被挤出来了

3）缓存也是分部在两个node中

统信软件
UNIONTECH

5.3.2 缓存工具分析iozone

已经系统的内存是32G，node0, node2上各有16G内存，2倍内存测试的文件大小是64G。由于系统本身占用了一部分内存及内存水位线的存在，系统最大只能缓存28G文件。

文件的前面40%，全都不在缓存里。而文件的最后40%，全部在缓存里。

前面40%都不在缓存，是因为文件比内存大，所以刚开始写文件时，由于文件比较小，所以文件都在缓存里，随着文件继续增大，当文件写到28G时，物理内存已经全部使用了。这时候再继续写文件时，由于缓存的释放基于先进先出的原理，因此会前面的缓存。所以当文件写完时，前面的缓存都已经释放了, 缓存全部集中在后面。

node0, node2上每个都占用一部分缓存是什么原因？ 因为iozone会随机在两个node上运行，在哪个node上运行就会申请哪个node上的缓存

如果是FIFO的回收原理，为什么缓存中第5行还有一点没有回收? 这个需要看多node情况的缓存回收机机制，多node情况下，缓存回收以node为单位来回收，每个node都启动了一个kswapd内核回收线程，它只负责回收自己node上的缓存。

假设回收之前node2上f[5](vmtouch统计结果中文件的第5行)的缓存占用为20%, f[6]的缓存占用为15.39%。存在这种场景： 当node0上的f[5]缓存回收完后，node2上的f[5]的缓存回收到图中的4.27%时不再收回缓存了，因为回收的缓存已经够了，也刚好结束了，那么此时看到的结果就是f[5]在node2上还有缓存。而导致这个结果的根本原因就是以node为单位的多线程回收机制。

统信软件 UNIONTECH

5.3.3 缓存工具分析iozone

iozone的测试原理是当写完成一个2倍物理内存大小的文件后，开始读这个2倍物理内存大小的文件。就是说写是一个测试项，读也是一个测试项。

由于文件大小比真正缓存大小的2倍还多，而缓存的回收又基于FIFO的机制来回收，这意味着写时缓存的28G内容在读时命中率几乎为0，因为读文件时从文件的开头开始读，但写时缓存的28G内容在文件的后半部分。此时的过程就是：

a)重新开始从磁盘读取数据，重新缓存，由于已经没有多余的内存空间，所以此时的缓存就是不断的回收已经缓存的28G文件后半部分内容，同时又缓存刚读进来的内容。

b)当读了文件前28G内容后，写时缓存的文件最后 28G内容已经全部被清空了，此时继续读文件28G之后的内容，此时的内容由于不在缓存中，继续读硬盘，假设读28G~29G的内容，那么文件0G~1G的缓存又被回收了，28G~29G的文件内容被缓存了。

c)继续读29G以后的内容，由于一直处理无多余内存，缓存一直被回收的状态，导致的结果缓存看不到命中的情况。别看缓存内容这么大，实际上是白忙活了，整个过程就是缓存不命中-> 回收旧缓存 --> 缓存刚读数据的过程， 属于缓存瞎忙的状态。

统信软件 UNIONTECH

5.4 iozone读性能优化

理清了iozone读数据和缓存的关系，有什么办法可以优化吗?

当前的问题主要是缓存无效，白忙活了，关闭缓存是不可能的，首先读写缓存的机制是linux默认的机制，只要读写不是direct方式，目前遇到的问题主要是因为文件的内容远超内存的大小。可以换一个角度理解成：iozone这种大文件缓存让缓存白缓存了，这是简单的FIFO机制面对两倍物理内存大小的文件读写的一种机制上的不足。

对于io读写，充分利用缓存是最能提高读写性能的办法，以这儿的iozone测试为例，让iozone写文件时如果能让缓存的28G缓存中有一部分是文件的前28G内容就好了，最好是如文件的前14G内容就缓存，如是是前28G内容，那么性能将是最好的，相对于硬盘，特别是低速的sat磁盘，缓存读花的时间几乎可以忽略，那么真正读磁盘的数据就是64G-28G=36G，性能提升为28G/64G=43.75%.

但这个是理想情况，几乎做不到，因为基于目前的回收机制不可能全部都不回收，因为后面的数据要读写。不过可以基于多node回收机制来思考，让28G内容中的大约14G在缓存里面，如假设node0不回收，node2回收。进程启动时通常都是在node0上运行，缓存也从node0的内存申请，一个node 16G内存，node0可用内存估计14G， 那么前14G的内存缓存的也就基本上文件的前14G内容了。

总结一下就是：优化思想是基于对内核的缓存机制的理解，再借助我们自己完善的vmtcouh工具找到具体的缓存情况后分析出来的，可见开发适合自己的缓存统计工具是多么重要。实际上我们还可以开发出文件缓存命中率统计功能，思想就是内核打点，用户态工具基于文件系统节点统计打印出来。

统信软件
UNIONTECH

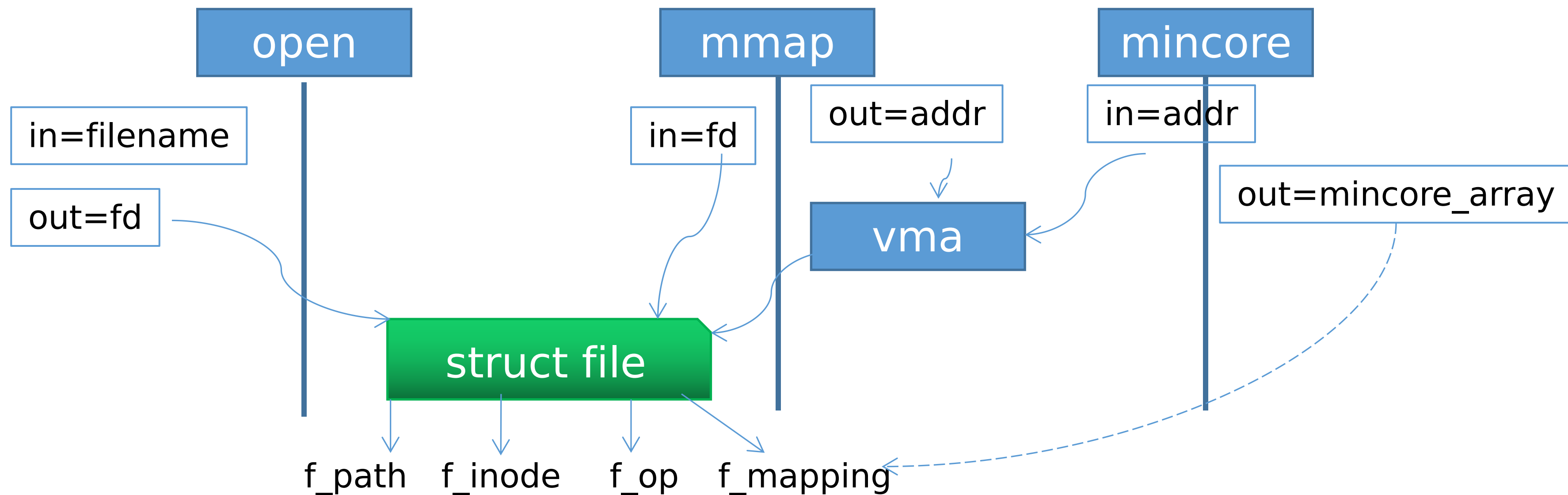
5.5 vmtouch实现关键技术介绍

这儿介绍关键技术是为了了解缓存的本质，即从用户态的角度来看怎样才在缓存里面呢？ 关键代码如下：

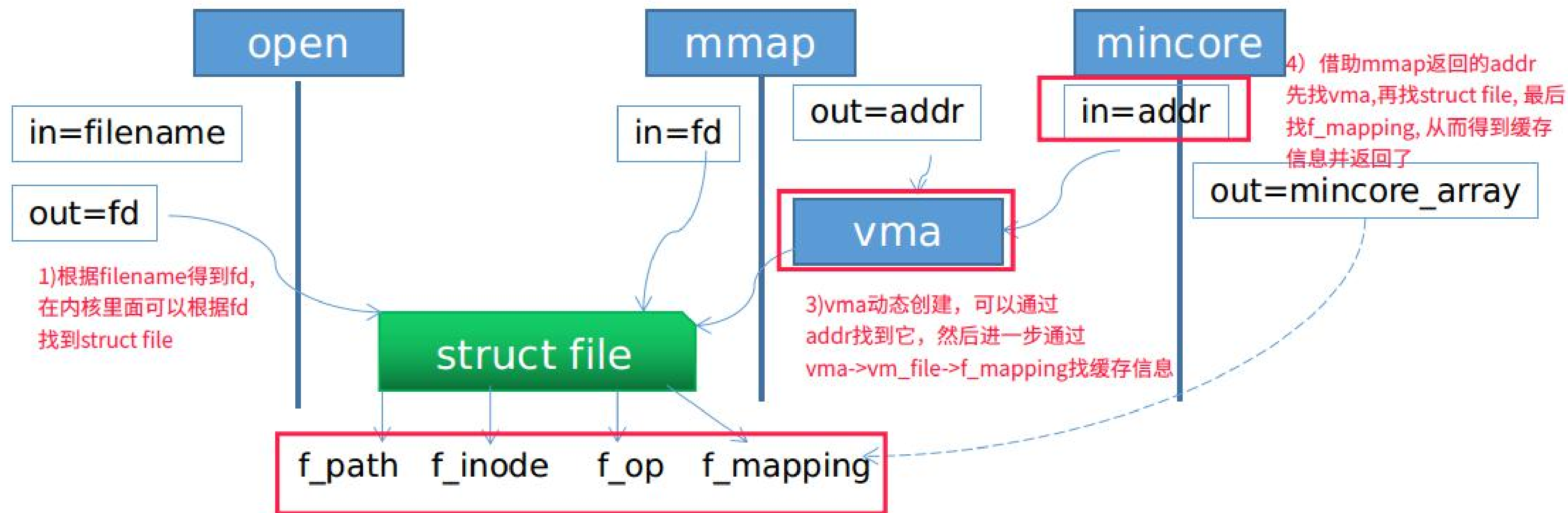
```
fd = open(path, open_flags, 0);
mem = mmap(NULL, len_of_range, PROT_READ, MAP_SHARED, fd, offset);
mincore(mem, len_of_range, (void*)mincore_array)
if (is_mincore_page_resident(mincore_array[i]))
    total_pages_in_core++;
```

统信软件 UNIONTECH

5.5.2 vmtouch实现关键技术介绍



5.5.3 vmtouch实现关键技术介绍



2)struct file下面的f_path, f_innode, f_op, f_mapping都是指针，
是文件共公部分，打开相同的文件时都会赋上相同的值。
其中f_mapping可以找到缓存页

统信软件 UNIONTECH

统信软件 UNIONTECH

5.5.4 vmtouch实现关键技术介绍

这儿介绍关键技术是为了了解缓存的本质，即从用户态的角度来看怎样才在缓存里面呢？ 关键代码如下：

fd = open(path, open_flags, 0);//fd指向的是要统计的缓存文件,除了下面的mmap要使用外，最关键的是通过fd在内核态新建struct file，这个struct file内部的f_path, f_inode, f_op, f_mapping等都是相同的。就是说只要共享相同的物理文件，那么无论这个文件打开多少次，虽然新创建了struct file这个内核数据结构，但是struct file内部的描述同一个物理文件的内容是相同的，而且这儿是f_path, f_inode, f_op, f_mapping都指向相同的地址。

mem = mmap(NULL, len_of_range, PROT_READ, MAP_SHARED, fd, offset);//调用map时，首先根据fd指open中打开的文件找出来，即内核态的struct file找出来。但是映射的地址和struct file没有关系，仅仅依赖进程的mmap_base,此时根据mmap_base偏移计算出来一个地址，当作虚拟地址返回给mem. 但是这儿不仅仅是表面上看到的返回mem虚拟地址这么简单，内核在内部还会调用mmap_region将mem所在的vma区和struct file关联起来，这样后面再用mem搜索到vma时，就可以基于vma->vm_file访问文件相关的内容了，如缓存情况。

mincore(mem, len_of_range, (void*)mincore_array)//mincore内部根据mem搜索到vma, 然后基于vma->vm_file->f_mapping就可以返回该文件的缓存情况了。所以内核保证通过vma访问同一个文件的f_mapping是得到缓存信息关键

if (is_mincore_page_resident(mincore_array[i]))//统计缓存

total_pages_in_core++;

统信软件
UNIONTECH

5.6 总结

1）缓存统计功能有很多，vmtouch是其中一种，我们在实战中加强了这个工具的功能，那就是将文件的缓存分布统计出来，将缓存的内存在哪个node上也统计出来。这个在大文件、多node服务器上很有用。

2）在单Node机器上(通常为桌面机器)，通常只需要统计这个文件有多少在缓存里面就可以了。但是对于多Node服务器场景，对于文件大小大于物理内存的场景，准备知道在Node上的缓存分布，文件本身的缓存分布，对性能优化至关重要。因为前者因为访问的内存缓存的不同性能有可能导致不可思议的性能差距，后者可让我们知道缓存是否真的有效(是否二次读取时缓存命中了)，这两点我们改时的vmtouchX工具都已经实现了。

3）工具有是否有效是根据需求来的，也许以后可能会面临新的需求，那么我们就要针对性的开发新的功能出来，我们需要具备的是根据需求灵活分析问题的能力，这些都需要在实战中磨炼出来。

统信软件
UNIONTECH

6. perf

统信软件
UNIONTECH

6.1 引言

perf是一个强大的性能分析工具，它可以统计系统运行过程中内核的一些指标参数，如

A)上下文切换次数、缺页次数、cpu指指令次数、分支预测成功次数、缓存命中率的相关统计等；

B)可以按热点函数进行排名，统计出频繁运行的函数，对这些热点函数进行调优；

C)可以反汇编正在运行的函数。

perf虽然提供了对这些信息的统计，但很难做到一看到这些参数就知道性能慢的原因. 如对于A)需要对内核的相关机制有比较准确地理解；对于Ｂ）如果统计的是内核态的函数还无法快速找到调用者，如假设正在执行数据copy的任务，此时perf给出的就是copy_page函数，但是这是一个非常底层的函数，到底是谁调用呢？此时需要进一步通过堆栈才确认，而且还需要排除干扰，因为调用copy_page的地方可以很多。

反汇编通常用在查看内核态动态生成的函数，内核有些函数使用了alternative机制，运行过程中根据内核配置或者芯片特性生成运行时代码，ftrace也是其中之一，ftrace相关配置打开后，内核编译时每个可跟踪的函数的头部都会加上一个nop指令，只有设置了需要跟踪此函数时会完成运行时指令的替换。

反汇编还有一个使用场景就是分析对手为什么更快，如假设已经定位到了copy函数比别人慢，那么用perf查看这个函数的反汇编代码是最快速的，不需要去反汇编整个内核，也不用担心静态反汇编代码和运行反汇编代码不一致的问题。

统信软件
UNIONTECH

6.2)理解perf中给出的缺页次数

perf中的缺页次数是最好理解的一个统计信息了，因为它和内存相关，本节通过一个简单例子来理解perf对此参数的统计。

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char** argv) {
  int i=0;
  int len=1024*1024*1024;
  char *psz=malloc(len);
  for (i = 0; i < len; i++)
        psz[i]=i;
  printf("psz[0 1024 len]=%d %d %d\n",psz[0], psz[1024], psz[len-1]);
  return 0;
}
```

申请1G的内存然后写确保会真的申请内存，最后加上一句打印确保for循环中赋值不被优化，如果被优化了就没有真正的内存申请了。

申请1G内存确保arm64 64K页机器上有机会申请512G大页内存

统信软件 UNIONTECH

6.2.2)理解perf中给出的缺页次数

调整内核大页模式为always模式，确保只要地址512G对齐了就会申请512G内存。

echo always >/sys/kernel/mm/transparent_hugepage/enabled

```
uos@uos-PC:~/zhoupeng/train/method6$ sudo perf stat ./allocmem
psz[0 1024 len]=0 0 255

 Performance counter stats for './allocmem':

          219.95 msec task-clock                #    0.999 CPUs utilized
               0      context-switches          #    0.000 K/sec
               0      cpu-migrations            #    0.000 K/sec
           8,219      page-faults               # 37529.680 M/sec
     571,863,368      cycles                    # 2611248.256 GHz
   1,016,377,848      instructions              #    1.78  insn per cycle
 <not supported>      branches
          50,053      branch-misses

     0.220136684 seconds time elapsed

     0.188117000 seconds user
     0.032020000 seconds sys
```

1)记住这个内存申请次数

2)指令并行度

3)整个时间为220ms

统信软件 UNIONTECH

6.2.3)理解perf中给出的缺页次数

调整内核大页模式为madvise模式，此时不会再申请512G大页内存。

echo madvise >/sys/kernel/mm/transparent_hugepage/enabled

```
uos@uos-PC:~/zhoupeng/train/method6$ sudo perf stat ./allocmem
psz[0 1024 len]=0 0 255

 Performance counter stats for './allocmem':

            226.74 msec task-clock                #    0.999 CPUs utilized
                 0      context-switches          #    0.000 K/sec
                 0      cpu-migrations            #    0.000 K/sec
            16,411      page-faults               # 72615.044 M/sec
       589,511,015      cycles                    # 2608455.819 GHz
     1,040,372,505      instructions              #    1.76  insn per cycle
   <not supported>      branches
            72,323      branch-misses

       0.226920922 seconds time elapsed

       0.198556000 seconds user
       0.028365000 seconds sys
```

1)内存申请次数大大增加

2)指令并行度保持不变

3)性能并大页内存时慢6ms,并不明显

统信软件 UNIONTECH

6.2.4)理解perf中给出的缺页次数

基于bpf确认内存申请情况

always模式下统计申请内存情况：

sudo trace-bpfcc -t '__alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order) "order=%d",order' -p 48616 > aa

```
2083 2.025945 48616    48616    allocmem           __alloc_pages_nodemask order=0
2084 2.025960 48616    48616    allocmem           __alloc_pages_nodemask order=0
2085 2.025976 48616    48616    allocmem           __alloc_pages_nodemask order=0
2086 2.025990 48616    48616    allocmem 1)记住2086 __alloc_pages_nodemask order=13  2)记住这个内存大小
2087 2.026078 48616    48616    allocmem           __alloc_pages_nodemask order=0
2088 2.132158 48616    48616    allocmem           __alloc_pages_nodemask order=0
2089 2.132182 48616    48616    allocmem           __alloc_pages_nodemask order=0
```

a)第2086次才申请内存，主要原因就是前面的地址还没有达到512G内存对齐

b)order=13，内存大小为64K*(2<<13)=512M

统信软件 UNIONTECH

6.2.5)理解perf中给出的缺页次数

基于bpf确认内存申请情况

madvise模式下统计申请内存情况：

sudo trace-bpfcc -t '__alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order) "order=%d",order' -p 49346 > bb

uos@uos-PC:~/src/kernel/arm-kernel-4$ cat bb|grep order=13

此时bb文件中再也找不到大内存的申请了,这也是perf工具中看到的缺页次数增加的原因。

统信软件 UNIONTECH

6.3)统计热点函数

```
#include <string.h>
void test1() {
        char acBuf[12];
     while(1) {
                memset(acBuf,0,sizeof(acBuf));
                memcpy(acBuf, "1234567890", 10);
         }
}
int main(int argc, char** argv) {
        test1();
    return 0;
}
```

1)test1是一个死循环函数

2)为了说明热点函数最终统计的是最后调用的那个函数，这儿有意在test1函数中调用memset和memcpy

统信软件 UNIONTECH

6.3.2)统计热点函数

gcc -g loopforever.c -o loopforever

./loopforever

sudo perf top

```
74.86%  libc-2.28.so      [.] __memcpy_generic
14.38%  libc-2.28.so      [.] __GI___memset_generic
 2.78%  loopforever       [.] memcpy@plt
 2.74%  loopforever       [.] memset@plt
 0.66%  [kernel]          [k] osq_lock
 0.27%  [kernel]          [k] finish_task_switch
 0.18%  [kernel]          [k] update_blocked_averages
 0.12%  [kernel]          [k] arch_cpu_idle
```

1)分别为memcpy和memset

热点函数集中在memcpy和memset, 但是看不到test1函数名。虽然由test1函数调用，但是一直调用的是memset和memcpy, 所以热点函数看不到test1.

统信软件 UNIONTECH

6.3.3)统计热点函数

将while(1)死循环中的memset和memcpy去掉，看看此时的热点函数

```
95.19%  loopforever     [.] test1
 0.61%  [kernel]        [k] osq_lock
 0.20%  [kernel]        [k] finish_task_switch
 0.15%  [kernel]        [k] update_blocked_averages
 0.10%  [kernel]        [k] arch_cpu_idle
 0.07%  [kernel]        [k] arch_counter_get_cntpct
```

while(1)死循环，没有调用memset和memcpy

此时的热点函数变成了test1, 因为test1函数里面只有一个while死循环语句，不再有函数的调用，这也说明perf是基于函数来统计的。

统信软件 UNIONTECH

6.4)理解指令执行个数

首先理解一下每周期指令执行个数

```
         16,412      page-faults          # 65386.454 M/sec
    654,438,592      cycles               # 2607325.068 GHz
  1,135,259,011      instructions         #    1.73  insn per cycle
<not supported>      branches
         77,876      branch-misses

    0.251932168 seconds time elapsed
```

1135259011/654438592.0=1.73,所以每周期指令指的就是在cycles这么多周期内执行的执令个数。

A)上图中cycles是运行的时钟周期个数，为654,438,592次，这么多时钟周期内执行了1,135,259,011条指令，那么平均每个周期就是1.73条指令。

B)对于同样的代码，这个值越大，性能就越好，最好理解的场景就是KP arm 和FT arm跑相同的代码，此值不一样。

对于同一个函数，不同的实现，在同一个芯片，谁的性能会更好呢？请看后面的例子分析。

C）后面都基于perf来分析，为了方便分析，在用户态实现了clear_page的汇编代码

测试环境为kp 64K arm server, ft 64K arm server

D)为了方便描述，每周期指令执行个数在后面都用**insn**表示

统信软件 UNIONTECH

6. 4 .2)汇编代码一

代码一：基于stnp指令清 0，但每次循环只调用一次stnp

```
.globl clear_page ; .align 2 ; clear_page:
  // .align  6
1:  stnp xzr, xzr, [x0]
    add     x0, x0, #0x10
    tst     x0, #0xfff
    b.ne    1b
    ret
.type clear_page, @function ; .size clear_page, .-clear_page
```

统信软件
UNIONTECH

6. 4 .3)汇编代码二

代码二：基于stnp指令清 0，但每次循环只调用两次stnp

```
.globl clear_page ; .align 2 ; clear_page:
  // .align  6
1:  stnp xzr, xzr, [x0]
    stnp xzr, xzr, [x0, #0x10]
    add    x0, x0, #0x20
    tst    x0, #0xfff
    b.ne   1b
    ret
.type clear_page, @function ; .size clear_page, .-clear_page
```

相对于代码一，每次循环中只调用两次stnp, 所以x0寄存器每次也只增加32B

统信软件 UNIONTECH

6. 4 .4)汇编代码三

代码三：基于stnp指令清 0

```
.globl clear_page ; .align 2 ; clear_page:
  // .align  6
1:  stnp xzr, xzr, [x0]
    stnp xzr, xzr, [x0, #0x10]
    stnp xzr, xzr, [x0, #0x20]
    stnp xzr, xzr, [x0, #0x30]
    add    x0, x0, #0x40
    tst    x0, #0xfff
    b.ne   1b
    ret
.type clear_page, @function ; .size clear_page, .-clear_page
```

统信软件 UNIONTECH

6. 4 .4)汇编代码四

代码四：基于stnp指令清 0，但每次循环只调用一次stnp，但计算器每次增加64B

```
.globl clear_page ; .align 2 ; clear_page:
  // .align  6
1:  stnp xzr, xzr, [x0]
    add    x0, x0, #0x40
    tst    x0, #0xfff
    b.ne   1b
    ret
.type clear_page, @function ; .size clear_page, .-clear_page
```

统信软件
UNIONTECH

6. 4 .5)汇编代码五

代码五：基于dc zva指令清 0

```
.globl clear_page ; .align 2 ; clear_page:
1:  dc  zva, x0
    add x0, x0, 64
    tst x0, #0xfff
    b.ne    1b
    ret
.type clear_page, @function ; .size clear_page, .-clear_page
```

统信软件
UNIONTECH

6. 4 .6)clear_page实现

调用clear_page的测试代码：

```
char __attribute__((aligned(64 * 1024)))a[1024 * 1024 * 1024] = {0};
int main ()
{
    int i,n;
    int64_t time1;    int64_t time2;    int64_t time3;
    time1 = GetUs();
    for (int i = 0; i < 1024 * 1024 * 1024; i++){//保证能申请内存出来
      a[i] = i;
    }
    time2 = GetUs();
    for (int j = 0;j<16*1024;j++){
      stnp_test(&a[64*1024*j]);/调用clear_page函数测试
    }
    time3 = GetUs();
    printf("allocmem(us)=%ld clearpage(us) = %ld\r\n", (time2 - time1), time3-time2);/打印申请内存和clear_page花的时间
    return 0;
}
```

统信软件
UNIONTECH

6.4.7)测试代码一

对比一：使用代码一在KP和FT上运行

```
uos@uos-PC:~/zhoupeng/clear_page$ sudo perf stat ./yzytest_stnp_o
allocmem(us)=222817 clearpage(us) = 31207

 Performance counter stats for './yzytest_stnp_o':

          257.72 msec task-clock                #    0.999 CPUs utilized
               0      context-switches          #    0.000 K/sec
               0      cpu-migrations            #    0.000 K/sec
          16,413      page-faults               # 63863.813 M/sec
     670,056,885      cycles                    # 2607225.233 GHz
   1,309,753,826      instructions              #    1.95  insn per cycle
   <not supported>    branches
          99,026      branch-misses
```

1)KP上清除内存需要31ms左右

2)每个时间周期执行1.95条指令

```
uos@uos-PC:~/zhoupeng/clear_page$ sudo perf stat ./yzytest_stnp_o
allocmem(us)=333391 clearpage(us) = 100980

 Performance counter stats for './yzytest_stnp_o':

          440.79 msec task-clock                #    0.999 CPUs utilized
               1      context-switches          #    2.273 M/sec
               0      cpu-migrations            #    0.000 K/sec
          16,413      page-faults               # 37302.273 M/sec
     925,650,507      cycles                    # 2103751.152 GHz
   1,256,902,248      instructions              #    1.36  insn per cycle
   <not supported>    branches
         131,715      branch-misses
```

1)FT上大约需要100ms

2)每个时钟周期执行指令条数为1.36

```
1:  stnp xzr, xzr, [x0]
    add   x0, x0, #0x10
    tst   x0, #0xfff
    b.ne  1b
    ret
```

A)测试程序代码一样，KP上跑完运行了13亿条指令，FT上跑完运行了12.56亿条指令；
B)对于clear_page这个函数本身，FT需要101ms左右，但是KP上只需要31ms.
巨大性能差距，初步估计FT的stnp比KP慢太多

统信软件 UNIONTECH

6.4.8)测试代码二

对比二：使用代码二在KP和FT上运行

```
allocmem(us)=222769 clearpage(us) = 29219

 Performance counter stats for './yzytest_stnp_o':

          256.32 msec task-clock                #    0.999 CPUs utilized
               1      context-switches          #    3.906 M/sec
               0      cpu-migrations            #    0.000 K/sec
          16,413      page-faults               # 64113.281 M/sec
     666,409,914      cycles                    # 2603163.727 GHz
   1,208,160,198      instructions              #    1.81  insn per cycle
 <not supported>      branches
          81,754      branch-misses
```

1）相对于只有一条stnp，快了1ms

2)每周期指令有下降，说明stnp本身并行度有提升，但因为执行指令减少，导致这个值下降

```
uos@uos-PC:~/zhoupeng/clear_page$ sudo perf stat ./yzytest_stnp_o
allocmem(us)=330273 clearpage(us) = 101433

 Performance counter stats for './yzytest_stnp_o':

          440.10 msec task-clock                #    0.999 CPUs utilized
               0      context-switches          #    0.000 K/sec
               0      cpu-migrations            #    0.000 K/sec
          16,413      page-faults               # 37302.273 M/sec
     924,192,804      cycles                    # 2100438.191 GHz
   1,157,173,875      instructions              #    1.25  insn per cycle
 <not supported>      branches
         116,693      branch-misses
```

1)没有得升，略有下降

2)该指标也下降了

```
1:  stnp xzr, xzr, [x0]
    stnp xzr, xzr, [x0, #0x10]
    add   x0, x0, #0x20
    tst   x0, #0xfff
    b.ne  1b
    ret
```

A)stnp变成两条后， insn都有所下降，下降的主要原因是clear_apge中的执行指令变少了，因为循环次数小降了１倍，除stnp指令外，其它的指令都减少了一半。

B)仔细观察发现KP和FT都下降了１亿条左右。但KP性能有微小的提升，FT则有微小下降。

有一种可能就是FT上的stnp几乎没有并行运行，KP上有一点并行运行，如KP上从一个时钟周期运行一条stnp变成了1.1条stnp, 不然是不会有性能提升的。

同时stnp从１条变成２条性能几乎没有变化

6.4.9)测试代码三

对比三：使用代码三在KP和FT上运行

```
allocmem(us)=222986 clearpage(us) = 29236
```
1)和两条stnp指令性能差不多

```
 Performance counter stats for './yzytest_stnp_o':

          256.71 msec task-clock                #    0.999 CPUs utilized
               1      context-switches          #    3.906 M/sec
               0      cpu-migrations            #    0.000 K/sec
          16,413      page-faults               # 64113.281 M/sec
     667,420,526      cycles                    # 2607111.430 GHz
   1,157,779,504      instructions              #    1.73  insn per cycle
   <not supported>    branches
          83,032      branch-misses
```
2)进一步下降，请看后面分析

```
uos@uos-PC:~/zhoupeng/clear_page$ sudo perf stat ./yzytest_stnp_o
allocmem(us)=329504 clearpage(us) = 103316
```
1)性能进一步下降

```
 Performance counter stats for './yzytest_stnp_o':

          440.85 msec task-clock                #    0.999 CPUs utilized
               0      context-switches          #    0.000 K/sec
               0      cpu-migrations            #    0.000 K/sec
          16,413      page-faults               # 37302.273 M/sec
     925,784,033      cycles                    # 2104054.620 GHz
   1,107,751,871      instructions              #    1.20  insn per cycle
   <not supported>    branches
         101,271      branch-misses
```
2)也下降了，请看后面分析

```
1:  stnp xzr, xzr, [x0]
    stnp xzr, xzr, [x0, #0x10]
    stnp xzr, xzr, [x0, #0x20]
    stnp xzr, xzr, [x0, #0x30]
    add   x0, x0, #0x40
    tst   x0, #0xfff
    b.ne  1b
    ret
```

A)Insn进一步下降

B)stnp从1条变成了４条并没有带来性能的提升

统信软件 UNIONTECH

6.4.10)测试代码四

对比四：使用代码四在KP和FT上运行

```
uos@uos-PC:~/zhoupeng/clear_page$ sudo perf stat ./yzytest_stnp_o
allocmem(us)=223165 clearpage(us) = 96232   1)KP机器，stnp的指令减少到以前的1//4,性能反而下降了，值得分析

 Performance counter stats for './yzytest_stnp_o':

            324.07 msec task-clock                #    0.999 CPUs utilized
                 0      context-switches          #    0.000 K/sec
                 0      cpu-migrations            #    0.000 K/sec
            16,413      page-faults               # 50657.407 M/sec
       842,573,448      cycles                    # 2600535.333 GHz
     1,107,368,588      instructions              #    1.31  insn per cycle
   <not supported>      branches
            81,200      branch-misses
```

```
1:  stnp xzr, xzr, [x0]
    add   x0, x0, #0x40
    tst   x0, #0xfff
    b.ne  1b
    ret
```

A)stnp只运行一条，但是for循环人为减少到以前的1/4,但是性能没有提升反而大幅下降

B）下降的肯定是stnp变慢了，变慢的原因很有可能是stnp从访问缓存变成了访问内存。因为之前的连续内存访问时可能一直保持高缓存利用率，缓存预涓一直在生效

统信软件 UNIONTECH

6.4.11)测试代码一和代码四

对比此时的缓存命中率

sudo perf stat -e cache-references -e cache-misses ./test_o

```
uos@uos-PC:~/zhoupeng/clear_page$ sudo perf stat  -e cache-references -e cache-misses ./yzytest_stnp_o
allocmem(us)=222935 clearpage(us) = 29176

 Performance counter stats for './yzytest_stnp_o':

       218,306,581      cache-references
           350,656      cache-misses              #    0.161 % of all cache refs
```

```
uos@uos-PC:~/zhoupeng/clear_page$ sudo perf stat  -e cache-references -e cache-misses ./yzytest_stnp_o
allocmem(us)=223015 clearpage(us) = 94587

 Performance counter stats for './yzytest_stnp_o':

       168,004,491      cache-references
        12,912,017      cache-misses              #    7.686 % of all cache refs
```

A)上图为代码一的缓存命中率，下图为代码四的缓存命中率。
可以看到代码四虽然clear_page的数量降到了代码一的1/4, 但是性能反而慢了很多。从30ms下降到 9 5ms左右。

导致性能下降的主要原因就是缓存命中率大大下降了

统信软件 UNIONTECH

6.5)总结

A)现在来分析从代码二变为代码一，下降了多少条指令

对于代码一：外循环16*1024次，clear_page内部有64*1024/16=4*1024次。

所以总共有16*1024*4*1024=64*1024*1024次。

当从代码一变为代码二时，clear_page的循环次数减少一半，此时减少的指令是add, tst,b.ne

代码一中运行了64*1024*1024*3=201326592次，减少一半就是减少到100663296,

所以大约减少了１亿条指令

B)再看看下降的时钟周期

KP上下降了670056885-666409914=3646971=364W个时钟周期

FT上下降了925650507-924192804=1457703＝164W表时钟周期

减少的是相同的代码，但KP上下降得更多，这儿有两种理解，一个是

add,tst, bl这些指令在KP上占用的时钟周期更多;

还有一种理解是stnp在KP上有一定的并行度，这个并行度导致小了200W表个时钟周期。

我没得第二个理解更有可能。如果这个假设方向正确，那么此时的并行度就是

200W/(64*1024*1024)=0.03, 即KP上从一条stnp变成两条stnp时仅仅只提高３%的并行度

统信软件
UNIONTECH

6.5.2)总结

C)不要指望增加stnp的并行个数能提高并行度，从kp上的测试来看，大约只提高了3%, FT还有略微下降。

D) 更好的性能提升是利用缓存预测和缓存命中率提升性能.

在连续的内存访问时反而更能触发cpu的缓存提前预读机制.

也就是说读写64K的数据反而比读写1/2, 1/4的数据快很多.

由于缓存的性能是内存的至少10倍以上，所以只有当读写数据下降到1/10左右时，此时两者的性能才接近.

E）insn并不是指指令并行度，它就是指一个时钟周期内运行指令的个数. 同样的代码不同的芯片时这个值越大，代表两个芯片的指令执行时间的性能差距. 如本测试上发现KP比FT快 2 倍，当然这个kp 的cpu缓存也有关. 因为stnp的性能提升信赖缓存.

F) insn >1时意味着一些指令执行很快. 同样的芯片不同的代码，并不能100%意味着insn越大此时的性能就越好，因为如果两者的运行时间一样，insn大的那个很有可能是运行了更多的指令导致. 如代码一和代码二对比，则时代码一运行了更多的add, bl, tst, 代码二中的stnp虽然并行度提高了3%, 即性能提高了3%，总时钟周期下降了一点，但是由于总的执行指令下降得更多，所以insn反而更低.

G)add, bl, tst这些令比stnp运行得快，可以估算一个时钟周期内的insn可以达到1.8以上.

H)由于本身不做cpu, 无法从cpu本身来准备度量，所以上面的总结都是定性分析.

统信软件
UNIONTECH

6.6)总结

A）当还对内核的相关细节没有那么了如指撑时，此时的perf还很有用，因为它可以提供很多指标给我们，如缺页次数，上下文切换次数，缓存命中率等等 。但任何指标都是两面的，如并不是缺页次数多性能就一定差，因为如果某个应用刚好是申请大页(512M)跨Node访问导致性能低下，那么更多的缺页次数反而性能更好，因为此时申请的小页内存(64K). 所以对指标的理解建立在对应用场景和内核的理解之上，不能固定思维去分析问题。

B）从我个人的角度来看，perf的缓存命中率统计、分支预测统计比较有用，因为内核目前还无法找到这些指标的直接跟踪，当然很有可能是我目前还没有了解这么深入，如perf是如何统计出来的呢?是不是可以先学学perf的统计然后去想办法自己灵性应用。是否有必要去这么研究取决于后面的需求，如某个场景的优化需要去看到底是什么引起cpu缓存命中率轻微下降了。

统信软件
UNIONTECH

7. 文件系统节点

统信软件
UNIONTECH

7.1)引言

一个应用除去业务逻辑以外的问题，最多的文题就是文件和内存相关的问题，如没有close的文件太多，导致打开socket失败；进程访问某个地址出现了异常，想知道这个地址是属于谁的？进程占用内存过多，想统计此进程占用内存详细信息等。

这些都可以基于内核提供的文件系统接中来得知，进程的相关信息都在/proc文件系统下面，假设进程的pid=10000, 那么该进程的所有信息都在/proc/10000下面，查看进程打开的文件，只需要查看/proc/10000/fd即可以，想要查找进程的内存地址分布，查看/proc/10000/maps文件内容即可，想要统计此进程的占用内存详细信息，如每个动态库到底占用了多少内存，mmap出来的内存空间到底有多少是申请内物理内存的，这些都可以基于/proc/10000/maps信息来二次开发。

除了fd和maps, 还有其它文件系统节点可以利用，如status 可以查看进程的状态等信息，statm可以查看进程的总内存占用信息。

本节主要举例讲解使用最常见的fd和maps, 如何基于它们来分析文件和内存相关的问题。

统信软件
UNIONTECH

7.2)使用fd分析问题

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
int main(int argc, char** argv) {
   int i =0;
   int fd;
   for (i = 0; i < 1024;i++) {
         fd = open("aa.txt", O_RDONLY);
         if (fd > 0)
            printf("fd[%d]=%d\n", i, fd);
         else
            printf("open file failed:%s\n", strerror(errno));
   }
   getchar();
   return 0;
}
```

统信软件
UNIONTECH

7.2.1)例子解释

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
int main(int argc, char** argv) {
  int i =0;
  int fd;
  for (i = 0; i < 1024;i++) {
      fd = open("aa.txt", O_RDONLY);
      if (fd > 0)
        printf("fd[%d]=%d\n", i, fd);
      else
        printf("open file failed:%s\n", strerror(errno));
  }
  getchar();
  return 0;
}
```

1）进程默认打开文件的次数最多只有1024次

2）当打开文件失败时，查看此时的错误信息

3)进程保护不退出，看此时的错误信息

统信软件 UNIONTECH

7.2.2)例子运行结果

```
fd[1018]=1021
fd[1019]=1022
fd[1020]=1023
open file failed:Too many open files
open file failed:Too many open files
open file failed:Too many open files
^[
```

1)第1024次打开文件时出现了错误，错误信息就是打开了太多的文件

```
uos@uos-PC:~/zhoupeng/train$ ls -l /proc/3338/fd
总用量 0
lrwx------ 1 uos uos 64 6月  24 22:05 0 -> /dev/pts/0
lrwx------ 1 uos uos 64 6月  24 22:05 1 -> /dev/pts/0
lrwx------ 1 uos uos 64 6月  24 22:05 10 -> /home/uos/zhoupeng/train/method12/aa.txt
lrwx------ 1 uos uos 64 6月  24 22:05 100 -> /home/uos/zhoupeng/train/method12/aa.txt
lrwx------ 1 uos uos 64 6月  24 22:05 1000 -> /home/uos/zhoupeng/train/method12/aa.txt
lrwx------ 1 uos uos 64 6月  24 22:05 1001 -> /home/uos/zhoupeng/train/method12/aa.txt
lrwx------ 1 uos uos 64 6月  24 22:05 1002 -> /home/uos/zhoupeng/train/method12/aa.txt
lrwx------ 1 uos uos 64 6月  24 22:05 1003 -> /home/uos/zhoupeng/train/method12/aa.txt
```

2)非常关键，基于ls -l 进程的文件夹查看文件打开信息

3)确实打开了大量的文件

总结：

当真实的环境出现了因为打开文件过多导致失败的例子时，此时进入/proc/xxx/fd去看是什么文件打开过多是非常容发现问题的.

像手机应用，打开socket及pipe是很常见的需求，如果因为异常导致很多文件没有关闭，导致打开socket失败，此时一定要先查看系统错误号errno得到内核给出的原因。

统信软件
UNIONTECH

7.3)使用fd分析例子二

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```
int main(int argc, char** argv) {
    int i =0;
    int fd;
    for (i = 0; i < 1020;i++) {
            fd = open("aa.txt", O_RDONLY);
            if (fd < 0)
                printf("open file:%d failed:%s\n", i, strerror(errno));
    }
    int sv[2]={0};
    int result = socketpair(AF_UNIX, SOCK_STREAM, 0, sv);
    if (result < 0) {
        printf("sv[0]=%d sv[1]= %d fd=%d %s\n", sv[0], sv[1], result, strerror(errno));
    }
    getchar();
    return 0;
}
```

统信软件
UNIONTECH

7.3.1)socket例子解释

```
int main(int argc, char** argv) {
  int i =0;
  int fd;
  for (i = 0; i < 1020;i++) {
      fd = open("aa.txt", O_RDONLY);
      if (fd < 0)
        printf("open file:%d failed:%s\n", i, strerror(errno));
  }
  int sv[2]={0};
  int result = socketpair(AF_UNIX, SOCK_STREAM, 0, sv);
  if (result < 0) {
    printf("sv[0]=%d sv[1]= %d fd=%d %s\n", sv[0], sv[1], result, strerror(errno));
  }
  getchar();
  return 0;
}
```

1)这儿打开1020个文件，在这之前打开了pts/0, pts/1, pts/2三个文件，总共是1023个文件

2)socketpair要打开两个文件，所以打开文件失败

统信软件 UNIONTECH

7.3.2)运行socket例子

```
./opensocket
sv[0]=0 sv[1]= 0 fd=-1 Too many open files


uos@uos-PC:~/zhoupeng/train$ ps -ef|grep opensocket
uos      3908  2131  0 23:11 pts/0    00:00:00 ./opensocket
uos      3911  2140  0 23:12 pts/1    00:00:00 grep opensocket
```

统计打开的文件个数

```
uos@uos-PC:~/zhoupeng/train$ ls -l /proc/3908/fd > aa3
uos@uos-PC:~/zhoupeng/train$ cat aa3|grep "uos"|wc -l
1023
```

结论:

已经打开了1023个文件，socketpari需要打开两个文件，所以打开文件失败了

统信软件
UNIONTECH

7.4)使用maps分析

```
#include <stdlib.h>
#include <sys/mman.h>
#define PAGE_SHIFT 16
#define PAGE_SIZE (1<<PAGE_SHIFT)

void *malloc(size_t size) {
   int pages = (size + PAGE_SIZE -1)/PAGE_SIZE+ 1;
   int len = pages << PAGE_SHIFT;
   char *addr = (char*)mmap(0, len, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, 0,0);
   mprotect(addr+len - PAGE_SIZE, PAGE_SIZE, PROT_NONE);
   char *retaddr= addr+(PAGE_SIZE-(size%PAGE_SIZE));
   return retaddr;
}
int main(int argc, char** argv) {
   char* mybuf = (char*)malloc(65539);
   mybuf[65538]=0;
   mybuf[65539]=0;
   return 0;
}
```

统信软件
UNIONTECH

7.4.1)maps例子解释

```
#include <stdlib.h>
#include <sys/mman.h>
#define PAGE_SHIFT 16
#define PAGE_SIZE (1<<PAGE_SHIFT)

void *malloc(size_t size) {            自己实现了一个malloc函数
   int pages = (size + PAGE_SIZE -1)/PAGE_SIZE+ 1;
   int len = pages << PAGE_SHIFT;
   char *addr = (char*)mmap(0, len, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, 0,0);
   mprotect(addr+len - PAGE_SIZE, PAGE_SIZE, PROT_NONE);
   char *retaddr= addr+(PAGE_SIZE-(size%PAGE_SIZE));
   return retaddr;
}
int main(int argc, char** argv) {
   char* mybuf = (char*)malloc(65539);    2)这儿调用了malloc函数，那么到底是调用glibc的还是自己的呢?
   mybuf[65538]=0;
   mybuf[65539]=0;
   return 0;
}
```

问题：到底使用谁的malloc函数呢？可以使用maps信息搞定

统信软件 UNIONTECH

7.4.2)运行maps例子

```
Breakpoint 1, main (argc=1, argv=0xfffffffe068) at memwrite.c:15
15          char* mybuf = (char*)malloc(65539);
(gdb) p malloc
$1 = {void *(size_t)} 0x400604 <malloc>
(gdb)
```

1) gdb打印出malloc函数的地址

```
uos@uos-PC:~/zhoupeng/train$ cat /proc/3949/maps
00400000-00410000 r-xp 00000000 08:04 743445675          /home/uos/zhoupeng/train/memwrite
00410000-00420000 rw-p 00000000 08:04 743445675          /home/uos/zhoupeng/train/memwrite
fffff7e00000-fffff7f60000 r-xp 00000000 08:03 201338164  /usr/lib/aarch64-linux-gnu/libc-2.28.so
fffff7f60000-fffff7f70000 rw-p 00150000 08:03 201338164  /usr/lib/aarch64-linux-gnu/libc-2.28.so
fffff7f70000-fffff7f80000 rw-p 00000000 00:00 0
fffff7fa0000-fffff7fb0000 r--p 00000000 00:00 0          [vvar]
fffff7fb0000-fffff7fc0000 r-xp 00000000 00:00 0          [vdso]
fffff7fc0000-fffff7fe0000 r-xp 00000000 08:03 201338160  /usr/lib/aarch64-linux-gnu/ld-2.28.so
fffff7fe0000-fffff7ff0000 r--p 00010000 08:03 201338160  /usr/lib/aarch64-linux-gnu/ld-2.28.so
```

2)地址在这个范围内，它属于memwrite进程本身，不是glibc的

总结：

通过maps给出的例子范围，可以很快得到答案。实际上gdb跟踪方式也可以快速得到，nm及readelf看符号地址方式也可以得到。但它们都有局限性。

对于gdb 需要跟踪进去运行才知道，如果是一个大系统一些函数不能直接运行或者是已经运行过了呢?

nm及readelf需要一个个去看，没有maps这个全局信息来得快，而且它不适合看动态库文件，因此它里面的符号需要重定位

统信软件
UNIONTECH

7.5)基于maps统计内存

从上节的maps内容可以知道，进程的maps包含了进程在用户态空间申请的所有内存。这种内存有如下两个特点：

i)这些内存可以栈、堆空间，也可以是匿名内存、共享内存，还可以是elf文件的代码段、数据段占用的内存。

ii)通过maps看到的只是用户地址空间，由于用户态内存基于页表缺页机制申请，因此很大部分的地址很有可能没有申请物理内存。

基于第一个特点来统计内存，可以完整地统计该进程在用户态空间占用的内存；基于第二个特点来统计内存，可以准确统计实际占用内存，如果做到基于每个vma地址来统计，那么就可以详细知道各个内存分布，可以准确地知道每个动态库、栈、堆、匿名的内存占用，可以方便分析问题，这个在android尤其有用，因为android无论是java虚拟机还是图片解码基于mmap申请的内存都是带名字的，有了名字就很容易知道申请的内存大小了，这个在分析内存泄露、内存优化时很有用。

maps只是给出了vma地址，要想知道真实的内存占用，还需要基于/proc/xxx/pagemap得到每个虚拟地址后面的物理地址，然后通过/proc/kpagecount知道物理地址的引用情况，如果进程独享(匿名内存)还是多个进程共享(动态库代码段)。

本节基于这些节点介绍如何实现这个强大的内存统计工具同时描述一下这些节点能够统计的原理。

统信软件
UNIONTECH

7.5.2)mmap申请内存代码

为了方便理解，先从简单例子开始，理解了例子的实现代码后再从内核角度出现讲解背后的原理。首先实现一个申请内存的进程，基于mmap申请

```
#include <sys/mman.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <string.h>
```

```
int main(int argc, char** argv) {
    long len=16*4096*128;
    char *myaddr= (char*)mmap(0, len, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, 0, 0);
    printf("pid=%d start=%ld end=%ld\n",getpid(), myaddr, myaddr+len);
      getchar();
      myaddr[0]=0;
       printf("myaddr[0]=0\n");
      getchar();
      printf("myaddr[65536*4]=0\n");
      myaddr[65536*4]=0;
       getchar();
        printf("myaddr[65536*8]=0\n");
        myaddr[65536*8]=0;
        getchar();
        return 0;
}
```

统信软件
UNIONTECH

7.5.2)mmap申请内存代码解释

为了方便理解，先从简单例子开始，理解了例子的实现代码后再从内核角度出现讲解背后的原理。

```
int main(int argc, char** argv) {
    long len=16*4096*128;
    char *myaddr= (char*)mmap(0, len, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, 0, 0);
    printf("pid=%d start=%ld end=%ld\n",getpid(), myaddr, myaddr+len);
    getchar();
    myaddr[0]=0;
    printf("myaddr[0]=0\n");
    getchar();
    printf("myaddr[65536*4]=0\n");
    myaddr[65536*4]=0;
    getchar();
    printf("myaddr[65536*8]=0\n");
    myaddr[65536*8]=0;
    getchar();
    return 0;
}
```

1）一页的大小为64K，申请128页物理内存

2)打印pid和映射出来的地址，方便后面统计

3)申请一页物理内存

4)再申请一页物理内存

5)再申请一页物理内存。注意申请的三页物理内存的虚拟地址是不连续的

统信软件 UNIONTECH 统信软件 UNIONTECH

7.5.3)统计内存代码

前面为申请内存进程代码，下面为统计进程内存代码

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

#define _BITS(x, offset, bits) (((x) >> (offset)) & ((1LL << (bits)) - 1))

#define PM_PAGEMAP_PFN(x)       (_BITS(x, 0, 55))
```

```
typedef unsigned long  uint64_t;
uint64_t get_pfn(uint64_t value) {
   return PM_PAGEMAP_PFN(value);
}
int get_page_count(int kpagecount_fd, uint64_t pfn, uint64_t *count_out) {
     int off;
     off = lseek(kpagecount_fd, pfn * sizeof(uint64_t), SEEK_SET);
     if (off < 0) return -1;
     off = read(kpagecount_fd, count_out, sizeof(uint64_t));
     return off;
}
```

统信软件
UNIONTECH

7.5.3)统计内存代码

```
int main(int argc, char** argv) {
   int i =0;
   int pagemap_fd;
   int pid=1;
   unsigned long start=0, end=0;
   unsigned long firstpage=0, numpages=0;
   int pagesize=64*1024;
   char filename[128];
   uint64_t *addr_range, off;
    int kpagecount_fd = open("/proc/kpagecount", O_RDONLY);
   if (argc < 4) {
         printf("procmem pid start end\n");
         return 0;
   }
   start = atol(argv[2]);
   end = atol(argv[3]);
   snprintf(filename, sizeof(filename),"%s%s%s", "/proc/",argv[1],
"/pagemap");
pagemap_fd = open(filename, O_RDONLY);
firstpage = start/pagesize;
numpages = (end-start)/pagesize;
addr_range = malloc(numpages * sizeof(uint64_t));
off = lseek(pagemap_fd, firstpage * sizeof(uint64_t), SEEK_SET);
int len = read(pagemap_fd, addr_range, numpages*sizeof(uint64_t));
for (i = 0; i < numpages; i++) {
            uint64_t count;
            if (addr_range[i] !=0) {
              get_page_count(kpagecount_fd, get_pfn(addr_range[i]), &count);
              printf("addr=0x%lx pfn=0x%lx count=%d\n", addr_range[i],
get_pfn(addr_range[i]), count);
            }
      }
   return 0;
}
```

统信软件
UNIONTECH

7.5.4)统计内存代码解释

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
```

1)按位取值

```c
#define _BITS(x, offset, bits) (((x) >> (offset)) & ((1LL << (bits)) - 1))

#define PM_PAGEMAP_PFN(x)      (_BITS(x, 0, 55))
```

2)取出55位(最多)物理地址

```c
typedef unsigned long  uint64_t;
uint64_t get_pfn(uint64_t value) {
    return PM_PAGEMAP_PFN(value);
}
```

3)实现一个函数可根据pagemap和值获取物理地址

```c
int get_page_count(int kpagecount_fd, uint64_t pfn, uint64_t *count_out) {
    int off;
    off = lseek(kpagecount_fd, pfn * sizeof(uint64_t), SEEK_SET);
    if (off < 0) return -1;
    off = read(kpagecount_fd, count_out, sizeof(uint64_t));
    return off;
}
```

4)读/proc/kpagecount文件读出该物理地址的引用次数。pfn为物理页编号，lseek按照物理页编号偏移，read读出的就是该物理页的引用次数了

统信软件 UNIONTECH

7.5.4)统计内存代码解释

```
int main(int argc, char** argv) {
  int i =0;
  int pagemap_fd;
  int pid=1;
  unsigned long start=0, end=0;
  unsigned long firstpage=0, numpages=0;
  int pagesize=64*1024;
  char filename[128];
  uint64_t *addr_range, off;
  int kpagecount_fd = open("/proc/kpagecount", O_RDONLY);
  if (argc < 4) {
      printf("procmem pid start end\n");
      return 0;
  }
  start = atol(argv[2]);
  end = atol(argv[3]);
  snprintf(filename, sizeof(filename),"%s%s%s", "/proc/",argv[1],
"/pagemap");
```

5)打开kpagecount文件

6)为了方便演示被统计进程的vma开始、结束地址直接传进来

```
  pagemap_fd = open(filename, O_RDONLY);
  firstpage = start/pagesize;
  numpages = (end-start)/pagesize;
  addr_range = malloc(numpages * sizeof(uint64_t));
  off = lseek(pagemap_fd, firstpage * sizeof(uint64_t), SEEK_SET);
  int len = read(pagemap_fd, addr_range, numpages*sizeof(uint64_t));
  for (i = 0; i < numpages; i++) {
      uint64_t count;
      if (addr_range[i] !=0) {
        get_page_count(kpagecount_fd, get_pfn(addr_range[i]), &count);
        printf("addr=0x%lx pfn=0x%lx count=%d\n", addr_range[i],
  get_pfn(addr_range[i]), count);
      }
   }
   return 0;
}
```

7)打开pagemap文件

8)得到开始页编号

9）得到需要获取页数

10)申请内存保存结果

11）先偏移到该虚拟地址对应的位置，然后读

12)addr_range[i]就是每页虚拟地址对应的物理地址，当然需要通过get_pfn将低55位物理地址获取出来，根据物理地址读kpagecount文件就可以将物理地址对应的物理页的引用次数读出来了。

这儿都是匿名内存，所以只要地址存在，那么count=1

统信软件 UNIONTECH

7.5.5)统计内存结果展示

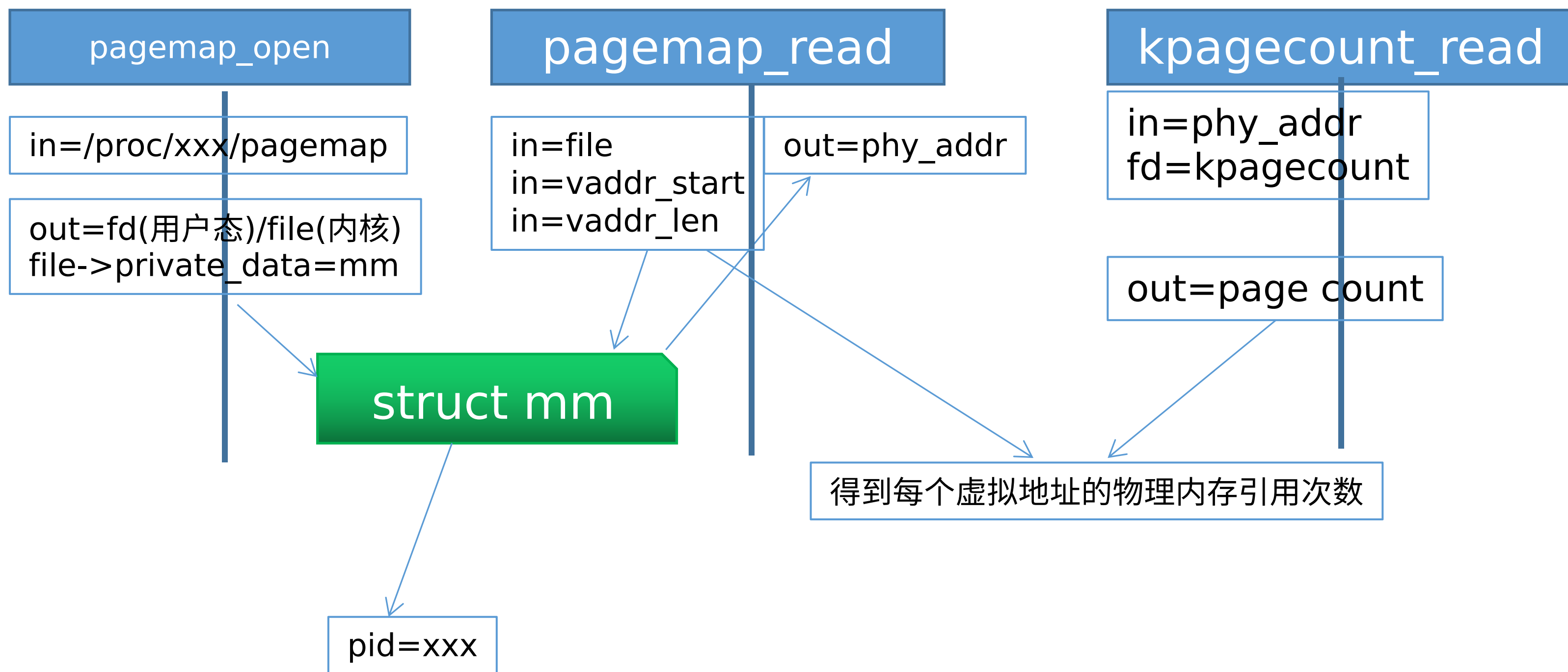
mmapmem进程申请内存，procmem统计mmapmem占用的内存，为了演示简单，mmapmem申请到的内存空间直接以参数形式输入给procmem. 真正的内存统计工具需要自己去读这个maps文件解析出来。很显然是可以实现的。

```
uos@uos-PC:~/zhoupeng/train/method12$ ./mmapmem
13533 281471393136640 281471401525248     1)刚开始只是申请了内存地址，没有申请内存

myaddr[0]=0     2)申请一页物理内存

myaddr[65536*4]=0     3)再申请一页物理内存

myaddr[65536*8]=0     4) 申请第三页物理内存
```

```
uos@uos-PC:~/zhoupeng/train/method12$ sudo ./procmem 13533 281471393136640 281471401525248     1)此时的结果是没有物理内存
uos@uos-PC:~/zhoupeng/train/method12$ sudo ./procmem 13533 281471393136640 281471401525248
addr=0x810000000023edc4 pfn=0x23edc4 count=1     2)此页物理内存被获取到了，引用计算为1,此进程独占
uos@uos-PC:~/zhoupeng/train/method12$ sudo ./procmem 13533 281471393136640 281471401525248
addr=0x810000000023edc4 pfn=0x23edc4 count=1     3)此时能获取到两页物理内存了，引用计数都为1
addr=0x8100000000240422 pfn=0x240422 count=1
uos@uos-PC:~/zhoupeng/train/method12$ sudo ./procmem 13533 281471393136640 281471401525248
addr=0x810000000023edc4 pfn=0x23edc4 count=1
addr=0x8100000000240422 pfn=0x240422 count=1     4) 此时能获取到4页物理内存了，引用计数都为1
addr=0x81000000002417cf pfn=0x2417cf count=1
uos@uos-PC:~/zhoupeng/train/method12$
```

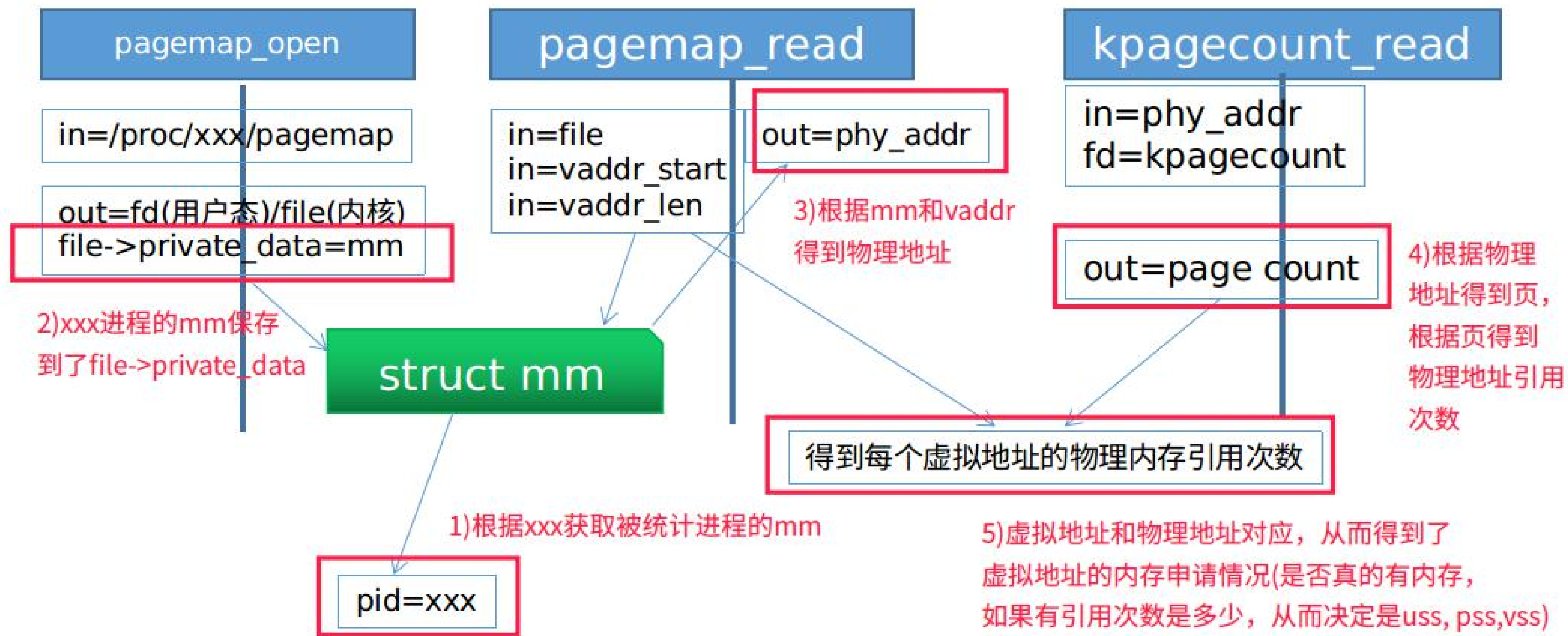
件
CH

7.6 内存统计实现关键技术介绍



统信软件 UNIONTECH

7.6 内存统计实现关键技术介绍



统信软件 UNIONTECH

7.6.1)pagemap_open理解

由于物理地址是通地读取pagemap文件得到的，因此看看打开文件和读文件时做了什么 。

```
static int pagemap_open(struct inode *inode, struct file *file)
{
    struct mm_struct *mm;

    mm = proc_mem_open(inode, PTRACE_MODE_READ);
    if (IS_ERR(mm))
        return PTR_ERR(mm);
    file->private_data = mm;
    return 0;
}
```

调用proc_mem_open时获取到了mm, 然后将mm赋值给了file->private_data
这个mm是被统计进程的mm, 如本例中的mmapmem进程。

统信软件 UNIONTECH

7.6.2)pagemap_read理解

```
static ssize_t pagemap_read(struct file *file, char __user *buf,
                size_t count, loff_t *ppos)
{
    struct mm_struct *mm = file->private_data;
    struct pagemapread pm;
    struct mm_walk pagemap_walk = {};
    unsigned long src;
    unsigned long svpfn;
    unsigned long start_vaddr;
    unsigned long end_vaddr;
    int ret = 0, copied = 0;

    if (!mm || !mmget_not_zero(mm))
        goto out;

    ret = -EINVAL;
    /* file position must be aligned */
    if ((*ppos % PM_ENTRY_BYTES) || (count % PM_ENTRY_BYTES))
        goto out_mm;
```

1）需要被统计的进程的mm

2)*ppos用户态传入，表示的是用户态虚拟地址，64位机器上，它必须是8字节对齐的；count表示读多少字节内容，读pagemap的本质是要将物理地址读出来所以假设要读n页，那么要读的内容长度就是8n了

上面获取到的mm就是pagemap_open函数设置进来的

统信软件
UNIONTECH

7.6.3)pagemap_read理解

```
/* do not disclose physical addresses: attack vector */
pm.show_pfn = file_ns_capable(file, &init_user_ns, CAP_SYS_ADMIN);

pm.len = (PAGEMAP_WALK_SIZE >> PAGE_SHIFT);
pm.buffer = kmalloc_array(pm.len, PM_ENTRY_BYTES, GFP_KERNEL);
ret = -ENOMEM;
if (!pm.buffer)
    goto out_mm;

pagemap_walk.pmd_entry = pagemap_pmd_range;
pagemap_walk.pte_hole = pagemap_pte_hole;
fdef CONFIG_HUGETLB_PAGE
pagemap_walk.hugetlb_entry = pagemap_hugetlb_range;
ndif
pagemap_walk.mm = mm;
pagemap_walk.private = &pm;

src = *ppos;
svpfn = src / PM_ENTRY_BYTES;
start_vaddr = svpfn << PAGE_SHIFT;
end_vaddr = mm->task_size;
```

1)pm.len=(1<<29)>>16=1<<13

2)申请的内存长度为8* (1<<13)=64K

3)设置需要遍历的回调函数

4) 指向被统计进程的mm

5)pm将是保存输出结果的地方

6)还原成用户成的虚拟地址。因为用户态传入时的值为*vaddr/pagesize)*sizeof(uint64)

上面的代码是pagemap_read函数的中间代码

统信软件 UNIONTECH

7.6.4)pagemap_read理解

```
ret = 0;
while (count && (start_vaddr < end_vaddr)) {
        int len;
        unsigned long end;

        pm.pos = 0;
        end = (start_vaddr + PAGEMAP_WALK_SIZE) & PAGEMAP_WALK_MASK;
        /* overflow ? */
        if (end < start_vaddr || end > end_vaddr)
                end = end_vaddr;
        ret = down_read_killable(&mm->mmap_sem);
        if (ret)
                goto out_free;
        ret = walk_page_range(start_vaddr, end, &pagemap_walk);
        up_read(&mm->mmap_sem);
        start_vaddr = end;

        len = min(count, PM_ENTRY_BYTES * pm.pos);
        if (copy_to_user(buf, pm.buffer, len)) {
                ret = -EFAULT;
                goto out_free;
        }
        copied += len;
        buf += len;
        count -= len;
}
*ppos += copied;
if (!ret || ret == PM_END_OF_BUFFER)
        ret = copied;
```

1)修正end地址，end地址最大为start_vaddr + (1<<29)，但不能大于mm->task_size.

然后这儿的虚拟地址也是被统计进程的，不过这儿的end实在太大，可以优后，后面写代码确认一下。这儿的目的是为了根据虚拟地址找到物理地址，找到了物理地址，再通过读kpagecount文件节点就可以读物理地址对应的页属性，从而可以知道这个物理地址的内存情况了。如是否独享，还是多个进程共享等。

3)pm.buffer是输出缓冲，len是要读的长度，它有可能比count小。所以这儿end地址应该进一步再end和start_vaddr+(count/8)*pagesize之间选一个更小的值

walk_page_range体现了基于页表根据虚拟地址获取物理地址这一本质，且它依赖页表pgd地址来至mm, 即被统计进程的mm

统信软件 UNIONTECH

7.7)kpagecount理解

和pagemap一样，需要看看kpagecount_read函数做了什么，open为通用函数，不用像pagemap做特别处理.

```
static ssize_t kpagecount_read(struct file *file, char __user *buf,
                               size_t count, loff_t *ppos)
{
    u64 __user *out = (u64 __user *)buf;
    struct page *ppage;
    unsigned long src = *ppos;
    unsigned long pfn;
    ssize_t ret = 0;
    u64 pcount;

    pfn = src / KPMSIZE;
    count = min_t(size_t, count, (max_pfn * KPMSIZE) - src);
    if (src & KPMMASK || count & KPMMASK)
        return -EINVAL;

    while (count > 0) {
        /*
         * TODO: ZONE_DEVICE support requires to identify
         * memmaps that were actually initialized.
         */
        ppage = pfn_to_online_page(pfn);

        if (!ppage || PageSlab(ppage))
            pcount = 0;
        else
            pcount = page_mapcount(ppage);

        if (put_user(pcount, out)) {
            ret = -EFAULT;
            break;
        }

        pfn++;
        out++;
        count -= KPMSIZE;
```

1）传入进来的参数除8后就是物理地址了

2）根据物理地址得到struct page

3)根据struct page得到页被引用次数，一次就是进程独占的内存了，否则很有可能是缓存、共享内存了。

3）传入的count=8 时，就只读一次了，所以可以一次读多个地址的，但是对用户口态比较难，因为不能确保物理地址是连续的，所以还是一页页读比较好

关键点是pfn得到

page.

page_mapcount

返回页引用次数

统信软件 UNIONTECH

7.8)总结

1)本节的重点是介绍如何基于内核提供的文件系统节点来分析问题，由于大多数程序都避免不了内存文件的使用，页存优化、内存改写、内存和文件泄露是程序员需要解决的常见问题，因此本节中的例子也都是内存及文件相关。

2)查看进程打开的文件，ls -l /proc/xxx/fd就可以了，灵活利用还可以查文件泄露问题

3)/proc/xxx/maps是一个非常常用的文件，用户态的内存访问地址都在此文件中，如果不在，说明用户态内存访问出现异常。

4)proc/xxx/pagemap, /proc/kpagecont是更高级别的使用，基于它们可以统计一个进程用户态详细的内存分别情况，为内存优化、内存泄露提供方向性的判断，它的作用比较类似我们改进的缓存统计工具来分析缓存类相关问题。

5)文件系统节点远远不止这几个节点，如ftrace就是基于文件系统节点控制运行的，因此基于文件系统节点的调式场景比这多得多，本文只是将最常用的拿来讲解抛砖引玉。

统信软件
UNIONTECH

8. 定制控制代码调试

统信软件
UNIONTECH

8.1)引言

虽然bpf 有强大的内核跟踪能力和性能统计能力，但是它只能统计大部分函数，对于参数信息也只能跟踪入口参数和函数的输出返回值，无法全面统计函数内部变量。

如果直接加上统计代码或者是打印代码也通常难以满足需求，因为没有准确的控制。如在通用函数里面增加统计会导致系统启动及运行过慢，由于没有条件控制或者是不好写条件控制代码，统计的性能很有可能不是自己需要的；对于打印代码，存在同样的问题，如假设值想统计某个进程的内存申请的详细信息，如果直接在内存申请函数里面打印，那么此时打印就是所有进程的内存申请了，如果控制pid的打印范围，由于被调试进程的pid是动态变化的，这个预先规定的打印范围要么是很难准确控制，要么是启动几次后就在控制范围之外了。

精确的控制方法就是基于全局变量的值当做条件变量来代码的运行，全局变量的值又基于文件系统来动态修改。

统信软件
UNIONTECH

8.2)基于/sys/kernel/profiling来控制

/sys/kernel/profiling是内核的文件系统节点，它对应的代码在kernel/ksysfs.c，由于内核的全局变量在内核相通的，因此所有的内核级别的代码都可以通过此全局变量来控制。

由于全局变量需要通过写控制，因此通过profiling_store函数来控制内核变量的值。

```
68 int g_debug_pid;
69 int g_debug_numa;
70 static ssize_t profiling_store(struct kobject *kobj,
71                       struct kobj_attribute *attr,
72                       const char *buf, size_t count)
73 {
74     int ret;
75     unsigned long cnt;
76
77     if (kstrtoul(buf, 0, &cnt)){
78         return -EINVAL;
79     }
80     if (cnt >=10000 && cnt <12000) {
81         g_debug_numa = cnt - 10000;
82         printk("g_debug_numa=%d\n", g_debug_numa);
83         return -EINVAL;
84     }
85     else if (cnt >=20000 && cnt < 100000) {
86         g_debug_pid = cnt - 20000;
87         printk("g_debug_pid=%d\n", g_debug_pid);
88         return -EINVAL;
89     }
```

1)增加需要控制代码的全局变量，g_debug_pid上精确到控制哪个进程
g_debug_numa表示控制numa相关代码的走向

2)将输入参数解析到long型变量cnt 中

3)让g_debug_numa的值在0~2000之间，这样基于与控制代码逻辑时可以用到1,2,4,8,16,....1024, 即可以控制10种不同的情况

4)返回参数错误，做到不要影响正常的内核流程

5)让进程的pid在0~80000之间，如果不满足条件可以继续扩大范围

8.2.2)基于/sys/kernel/profiling来控制

使用g_debug_numa控制变量的地方

```
3923 out:
3924     if (g_debug_numa & 1) {
3925         printk("dont invoke task_numa_fault, pid=%d page_nid=%d flags=0x%lx last_cpupid=%d\n",
3926                 current->pid, page_nid, flags, last_cpupid);
3927     }
3928     else {
3929         if (page_nid != -1)
3930             task_numa_fault(last_cpupid, page_nid, 1, flags);
3931     }
3932     return 0;
3933 }
```

1)人为地控制内核代码走向，如果此变量的值为1，那么就不执行下面的task_numa_fault,从而可以测试没有内存balance时的性能情况

统信软件 UNIONTECH

8.2.3)基于/sys/kernel/profiling来控制

使用g_debug_pid控制变量的地方

```
209     if (nsegs == 1 && seg_size > front_seg_size) {
210         front_seg_size = seg_size;
211         flag |= 4096;
212     }
213     bio->bi_seg_front_size = front_seg_size;
214     if (seg_size > bio->bi_seg_back_size) {
215         bio->bi_seg_back_size = seg_size;
216         flag |= 8192;
217     }
218     if(current->pid == g_debug_pid) {
219         if (g_debug_page & 128)
220         {
221             printk("flag=0x%x for_nrs=%d do_split=%d m_segs=%d m_secs=%d secs=%d nsegs=%d seg_size=%d b_size=%d cur_bvlen=%d cluster=%d m_seg
    _size=%d split_nrs=%d\n",
222                 flag, for_nrs, do_split,
223                 queue_max_segments(q), max_sectors, sectors, nsegs, seg_size,
224                 b_size, cur_bv_len,blk_queue_cluster(q), queue_max_segment_size(q), split_nrs);
225         }
226     }
```

1)自己增加的方便查看代码运行逻辑的变量

2)blk_bio_segment_split函数是一个对bio进行执行的函数，只关心特定的进程的读写io时的对bio的执行情况，如dd进程，iozone进程
下面都是要打印的内部变量，有些变量如flag还是自己增加，可以方便看代码的运行逻辑

8.2.4)基于/sys/kernel/profiling来控制

修改g_debug_numa控制变量的值

echo 10001 > /sys/kernel/profiling

修改g_debug_pid控制变量的值，假设iozone进程的pid是19679

echo 39679 > /sys/kernel/profiling

统信软件
UNIONTECH

9. 互换内核调试

统信软件
UNIONTECH

9.1)引言

该节的调试比较适合内核方面的问题分析，如两个系统性能差距很大，到底是内核的原因还是系统的原因呢？此时互换内核测试是一种比较有效的手段。

以uos和kylin为例，假设kylin某个特性的性能比uos快，如创建0K文件总是比uos快，把kylin内核放到uos系统上跑发现性能和uos一样，也很慢，这说明不是内核本身的问题，但也基本不可能是系统某个服务导致，因为服务影响文件创建的逻辑很难说通，但是系统某个配置或者启动参数中的某个配置影响内核是有可能的，到底是什么配置会有这么大威力呢？在大量的不同系统参数中可以找出来，但是需要花很长时间，有没有更快、更有把握的方法呢？

可以偿试将uos内核放到kylin系统上去跑，此时如果没有意外uos内核在kylin系统上也会跑得很快，那么此时用uos系统＋uos内核和kylin系统＋uos内核将是比较好的对比方案。因为前者很慢，后者很快。而uos内核我们是有源代码的，此时就可以用第8节的定制代码调试来分析问题，找到具体慢的点了，此时时某个配置导致某个函数慢，也有可能是某个新增功能导致某个函数慢，也有可能是某个参数不同导致某个函数慢。很显然这种情况找到了慢的点后再找背后的原因也就方便多了。

统信软件 UNIONTECH

9.2)麒麟内核在uos系统上运行

内核启动文件都在/boot下面，在uos系统上启动内核需要initrd 和vmlinz，而麒麟系统上是采用了uimage 和 initrd，这是两个系统启动内核的区别，其他文件都是可以不移植的，像config ,dtb 等等。但是具体我没有仔细测试过那些是必须要的，保险点就都复制过来。

	example
config	config-4.4.131-20210120.kylin.desktop-generic
initrd	initrd.img-4.4.131-20210120.kylin.desktop-generic
uimage	uImage-4.4.131-20210120.kylin.desktop-generic
vmlinuz	vmlinuz-4.4.131-20210120.kylin.desktop-generic
initramfs	initramfs.img-4.4.131-20210120.kylin.desktop-generic

复制内核文件后，内核启动过程中还需要加载模块，有些很重要的模块可能会影响系统运行，为了保证系统能正常启动，还需要复制麒麟的模块，在/lib/modules 会有一个’uname -r’目录，将这个目录复制到uos对应的路径就行。

统信软件
UNIONTECH

9.3)uos内核在麒麟系统上运行

麒麟系统上启动内核采用的是uimage 和initrd，但是我们系统上没有uimage，只有vmlinuz。但是都可以启动内核，因为vmlinuz 和uImage 都是压缩vmlinx得到的内核引导文件，只不过uImage 相对复杂，uImage是对zImage进行加工，在其首部加了一些内核信息（时间、日期、版本等等，信息长度为0x40），而zImage是压缩vmlinux时加了一段解压启动的代码，区别就是解压vmlinz 内核不会自己启动，还需要搭配boot 命令才能让内核启动，而uImage 不需要boot 命令，当然还有一些其他的技术细节。

```
uos@uos-N-A:~$ sudo file uImage-4.4.131-20210120.kylin.desktop-generic
uImage-4.4.131-20210120.kylin.desktop-generic: u-boot legacy uImage, Linux-4.4.131-20210120.kylin.desMZ, Linux/ OS Kernel Image (Not compressed), 38501376 bytes, Wed Jan 20 03:28:15 2021, Load Address: 0x80080000, Entry Point: 0x80080000, Header CRC: 0x8A30B122, Data CRC: 0x0C5E2C9C
uos@uos-N-A:~$ sudo file vmlinuz-4.4.131-20210120.kylin.desktop-generic
vmlinuz-4.4.131-20210120.kylin.desktop-generic: gzip compressed data, max compression, from Unix
uos@uos-N-A:~$
```

统信软件
UNIONTECH

9.3.2)uos内核在麒麟系统上运行

在麒麟系统上运行uos内核时需要手动修改grub文件，将uImage 和 initrd位的文件名换成uos 的vmlinz 和 initrd，最后加一行boot命令，用来启动vmlinz。当然别忘了将uos /lib/modules 下的模块复制过去。

替换前：

```
        search --no-floppy --fs-uuid --set=root 941e4e82-4ff3-4d70-a237-4a218916f496
        fi
        echo    'Loading Linux 4.4.131-20210120.kylin.desktop-generic ...'
        linux   /uImage-4.4.131-20210120.kylin.desktop-generic root=UUID=abf84759-1841-4555-94c3-582e345ae5d3 rw  qui
loglevel=0 resume=UUID=ad1c6e4d-ba75-4646-b4f6-ef165c7ed3a4 rootdelay=10 KEYBOARDTYPE=pc KEYTABLE=us security=  selinux=0 bac
        echo    'Loading initial ramdisk ...'
        initrd  /initrd.img-4.4.131-20210120.kylin.desktop-generic
    }
    menuentry 'Kylin V10, with Linux 4.4.131-20210120.kylin.desktop-generic (restore mode)' --class kylin --class gnu-lin
```

替换后：

```
        echo    'Loading Linux 4.4.131-20210120.kylin.desktop-generic ...'
        linux   /vmlinuz-4.19.0-arm64-desktop-ysj root=UUID=abf84759-1841-4555-94c3-582e345
security=
        echo    'Loading initial ramdisk ...'
        initrd  /initrd.img-4.19.0-arm64-desktop-ysj
        boot
```

统信软件
UNIONTECH

9.4)总结

A)麒麟内核在uos系统上运行

1、复制麒麟/boot 目录下相关文件到uos对应目录下

2、复制麒麟/lib/modules 下的目录到uos对应目录下

3、update-grub 一下重新生成grub，启动的时候选择麒麟内核就行。

B)uos内核在麒麟系统上运行

1、将uos系统上的/boot 目录下的文件复制到麒麟系统/boot下

2、修改麒麟的grub.cfg，这个配置文件有两个，一般麒麟用的是/boot/efi 下面的，通过grep可以找到。

3、复制uos的模块到麒麟系统上。

统信软件
UNIONTECH

10. 汇编调试

10.1)引言

没有源代码的二进库、c库、内核调试时有时汇编级代码调试必不可少，特别是解决死机型问题、理解内核层和cpu相关的汇编代码、剖析竞争对手的优化机制等。

统信软件
UNIONTECH

10.2)代码一 栈空间溢出调试

```
#include <stdlib.h>
#include <stdio.h>
void processBuf(long *lBuf, int len) {
   long sum=0;
      int i = 0;
      for (i = 0; i < len; i++) {
            sum += lBuf[i];
      }
      printf("sum=%ld\n", sum);
}
void calc_array(int w, int h) {
      int i;
      int stackLen= w*h;
      long llBuf[stackLen];
      for (i = 0; i < stackLen; i++) {
            llBuf[i]=i*2;
      }
      processBuf(llBuf, stackLen);
}
```

```
int main(int argc, char** argv) {
      int w=800, h=600;
   if (argc > 2) {
            w = atoi(argv[1]);
            h = atoi(argv[2]);
      }
      calc_array(w,h);
   return 0;
}
```

统信软件 UNIONTECH

10.2.2)代码一　栈空间溢出调试

```c
#include <stdlib.h>
#include <stdio.h>
void processBuf(long *lBuf, int len) {
  long sum=0;
    int i = 0;
    for (i = 0; i < len; i++) {
        sum += lBuf[i];
    }
    printf("sum=%ld\n", sum);
}
void calc_array(int w, int h) {
    int i;
    int stackLen= w*h;
    long llBuf[stackLen];
    for (i = 0; i < stackLen; i++) {
        llBuf[i]=i*2;
    }
    processBuf(llBuf, stackLen);
}
```

6)for循环中完成计算，这样第4)点中的赋值也就不会被编译器优化了

3)w*h作为数组的长度，llBuf是局部变量，所以最终占用的是栈空间

4)开始给数组赋值，如果长度过大，最终会导致栈空间不足而出现段错误

5)开始处理前面数组的内容，如果没有这个函数的调用，-O3编译时上面的for循环会被优化掉

```c
int main(int argc, char** argv) {
    int w=800, h=600;
  if (argc > 2) {
        w = atoi(argv[1]);
        h = atoi(argv[2]);
    }
    calc_array(w,h);
  return 0;
}
```

1)默认的w=800,h=600 可以通过输入参数改变，如 w=1920, h=1080

2)w,h作为参数传入

统信软件 UNIONTECH

统信软件 UNIONTECH

10.2.3)代码一 栈空间溢出调试

1)首先直接运行，看到的是如下错误

./teststack 1920 1080

段错误

2)gdb运行

```
Reading symbols from ./teststack...(no debugging symbols found)...done.
(gdb) set args 1920 1080
(gdb) r
Starting program: /home/uos/zhoupeng/teststack/teststack 1920 1080

Program received signal SIGSEGV, Segmentation fault.
0x000000000040074c in calc_array ()
(gdb)
```

1)设置参数

2)这儿出现了段错误

统信软件 UNIONTECH

10.2.4)代码— 栈空间溢出调试

3)gdb反汇编

由于出现段错误的地址是0x40074c，因此反汇编400740~400750这段地址

```
(gdb) disas 0x0000000000400740,0x0000000000400750
Dump of assembler code from 0x400740 to 0x400750:
   0x0000000000400740 <calc_array+96>:  add     v1.4s, v1.4s, v3.4s
   0x0000000000400744 <calc_array+100>: sxtl    v2.2d, v0.2s
   0x0000000000400748 <calc_array+104>: sxtl2   v0.2d, v0.4s
=> 0x000000000040074c <calc_array+108>: stur    q2, [x2, #-32]
End of assembler dump.
```

stur代表写, 写的地址是$x2-32, 直接用gdb的内存查看命令x

```
(gdb) x $x2-32
0xffffff02bdd0: Cannot access memory at address 0xffffff02bdd0
(gdb) x $x2
0xffffff02bdf0: Cannot access memory at address 0xffffff02bdf0
```

$x2=0xffffff02bdd0, 这个地址是不可访问的

统信软件
UNIONTECH

10.2.5)代码一 栈空间溢出调试

4) 跟踪x2寄存器的赋值

由于程序死在calc_array函数，因此对calc_array函数设置断点，且单步执行过程中查看x2寄存器中的值何时发生变化。

b calc_array

display /x $x2

设置好后按r运行，当运行到calc_array函数时，按si单步跟踪执行

```
0x0000000000400724 in calc_array ()
1: /x $x2 = 0xa
(gdb)
0x0000000000400728 in calc_array ()
1: /x $x2 = 0xa
(gdb)
0x000000000040072c in calc_array ()
1: /x $x2 = 0x400000
(gdb)
0x0000000000400730 in calc_array ()
1: /x $x2 = 0x400000
(gdb)
0x0000000000400734 in calc_array ()
1: /x $x2 = 0x400000
(gdb)
0x0000000000400738 in calc_array ()
1: /x $x2 = 0xffffff02bdd0
```

1)初始值为0xa

2)在40072c时变成了0x400000

3)在400738这条指令时变成了异常值

总结：

x2寄存器的值在0x40072c时，从0xa变成了0x400000

在0x400738时，从0x400000变成了0xffffff02bdd0

现在需要看看这两个地址的值是如何变化的

统信软件 UNIONTECH

10.2.6)代码一 栈空间溢出调试

4) 跟踪x2寄存器的赋值

反汇编

disas 0x0000000000400728,0x0000000000400738

```
0x0000000000400734 in calc_array ()
1: /x $x2 = 0x400000
(gdb)
0x0000000000400738 in calc_array ()
1: /x $x2 = 0xffffff02bdd0
(gdb) p /x $x2
$8 = 0xffffff02bdd0
(gdb) disas 0x0000000000400728,0x0000000000400738
Dump of assembler code from 0x400728 to 0x400738:
   0x0000000000400728 <calc_array+72>:  adrp    x2, 0x400000
   0x000000000040072c <calc_array+76>:  umaddl  x1, w1, w5, x3
   0x0000000000400730 <calc_array+80>:  ldr     q1, [x2, #2352]
   0x0000000000400734 <calc_array+84>:  mov     x2, x3
End of assembler dump.
```

结论：需要进一步查看x3的值是如何来的

1)x2的值首次赋值为0x400000

2)x3赋值给x2, x3已经是0xffffff02bdd0了

统信软件 UNIONTECH

10.2.7)代码一 栈空间溢出调试

5) 跟踪x3寄存器的赋值

b calc_array

display /x $x3

si单步跟踪执行

```
0x0000000000400714 in calc_array ()
2: /x $x3 = 0xffffffffe1e6
(gdb)
0x0000000000400718 in calc_array ()
2: /x $x3 = 0xffffff02bdd0
(gdb) disas 0x0000000000400714,0x0000000000400718
Dump of assembler code from 0x400714 to 0x400718:
   0x0000000000400714 <calc_array+52>:  lsl     x3, x4, #3
End of assembler dump.
(gdb) disas 0x0000000000400704,0x0000000000400718
Dump of assembler code from 0x400704 to 0x400718:
   0x0000000000400704 <calc_array+36>:  mov     x1, sp
   0x0000000000400708 <calc_array+40>:  sub     w6, w0, #0x1
   0x000000000040070c <calc_array+44>:  cmp     w6, #0x3
   0x0000000000400710 <calc_array+48>:  lsr     x4, x1, #3
   0x0000000000400714 <calc_array+52>:  lsl     x3, x4, #3
End of assembler dump.
```

1)400718这条指令发生了变化

2)反汇编400718地址之前的代码

3)x3由x4决定
x4由x1决定
x1就是sp的值

(gdb) p /x $sp
$10 = 0xffffff02bdd0
(gdb) x $sp
0xffffff02bdd0: Cannot access memory at address 0xffffff02bdd0
结论：
sp的值就已经越界了，所以继续查看sp寄存器的值

统信软件 UNIONTECH

10.2.8)代码一 栈空间溢出调试

6) 跟踪sp寄存器的赋值

b calc_array

display /x $x3

si单步跟踪执行

```
0x00000000004006fc in calc_array ()
3: /x $sp = 0xffffffffddd0
(gdb)
0x0000000000400700 in calc_array ()
3: /x $sp = 0xffffff02bdd0
(gdb) x $sp
0xffffff02bdd0: Cannot access memory at address 0xffffff02bdd0
(gdb) disas 0x00000000004006f0,0x0000000000400700
Dump of assembler code from 0x4006f0 to 0x400700:
   0x00000000004006f0 <calc_array+16>:   cmp     w0, #0x0
   0x00000000004006f4 <calc_array+20>:   add     x1, x1, #0xf
   0x00000000004006f8 <calc_array+24>:   and     x1, x1, #0xfffffffffffffff0
   0x00000000004006fc <calc_array+28>:   sub     sp, sp, x1
End of assembler dump.
(gdb) p /x $x1
$12 = 0xfd2000
```

1)在400700处发生了变化

2)反汇编查看发生变化的原因

3)减去了x1导致

结论：跟踪x1寄存器的值

统信软件 UNIONTECH

10.2.9)代码一 栈空间溢出调试

7) 跟踪x1寄存器的赋值

b calc_array

display /x $x1

si单步跟踪执行

```
(gdb) disas 0x00000000004006e0,0x00000000004006f4
Dump of assembler code from 0x4006e0 to 0x4006f4:
   0x00000000004006e0 <calc_array+0>:   mul     w0, w0, w1
   0x00000000004006e4 <calc_array+4>:   stp     x29, x30, [sp, #-16]!
   0x00000000004006e8 <calc_array+8>:   mov     x29, sp
   0x00000000004006ec <calc_array+12>:  sbfiz   x1, x0, #3, #32
=> 0x00000000004006f0 <calc_array+16>:  cmp     w0, #0x0
End of assembler dump.
(gdb) p /x $x0
$19 = 0x1fa400
(gdb) p  $x0
$20 = 2073600
(gdb) p 1920*1080
$21 = 2073600
(gdb) p /x $x0<<3
$22 = 0xfd2000
(gdb) p /x $x1
$23 = 0xfd2000
(gdb)
```

结论：根据输入参数来决定数组的大小导致了堆栈overflow

1)x0就是输入参数的值x0=1920*1080

2)x0<<3位的值赋值给了x1,可以想像是因为x0是一个长度，x1指向的是一个如double or long的数组

10.2.10)代码一 栈空间溢出调试

8)调整堆栈大小验证问题的正确性

ulimit -s unlimited

结果OK

进一步将堆栈大小设置为16M

ulimit -s 16384

结果也OK

```
void calc_array(int w, int h) {
    int i;
    int stackLen= w*h;
    long llBuf[stackLen];
    for (i = 0; i < stackLen; i++) {
        llBuf[i]=i*2;
    }
    processBuf(llBuf, stackLen);
}
```

1)申请了大内存堆栈，也就是汇编中发现的x1值过大

2)一写就会出现栈空间溢出问题

统信软件 UNIONTECH

10.3)总结

A) 汇编调用在GDB一节中用到了，perf中也用到了，实际上遇到比较难的问题有可能用排除法时就能找到原因；理解原理后试错法也有可能找到原因，如这儿的直接将堆栈限制调大就能解决，从而快速确定原因。但是汇编跟踪能够让我们知道后面的本质，知道了后面的本质就更容易举一反三。

B)对于本节中的例子，栈空间overflow的本质是SP指针overflow, 需要去扩展空间，但是内核不给扩展了(因为8M限制)，相当于不能申请内存，最后出现写异常，抛出段错误异常。

C）汇编调试通常都是分析一些比较常识性的问题，如 GDB一节中的动态库符号重定位，这儿的栈空间溢出，我们还可以分要局部变量初始化（为什么MIPS不初始化时的变量值是1, arm上不初始化时是0），这就是汇编调试的魅力，让我们知道本质。

D）汇编调试还有一个有用场景就是内核学习之初，一些汇编代码不好理解时及不好性能测试时，可以在用户态写一个基于GDB理解清楚。

统信软件
UNIONTECH

10.3)内核栈自动增长机制研究

在对内核栈处理还不态清楚的情况下，找什么样的入口点来来分析呢?

由于栈空间内存没有显式内存申请函数，因此内核从 RLIMIT_STACK 这个宏的使用开始分析

先搜索这个宏的定义

grep -rns --include=*.h "RLIMIT_STACK" .

发现它在./include/uapi/asm-generic/resource.h:19

:#define RLIMIT_STACK 3 /* max stack size */

再搜索内核使用它的地方:

grep -rns --include=*.c "RLIMIT_STACK" .

结果发现了mmap.c中的这行代码，

./mm/mmap.c:2289:   if (size > rlimit(RLIMIT_STACK))

这行代码对应的函数就是acct_stack_growth

统信软件
UNIONTECH

10.3.2)基于bpf快速确认此函数的调用情况

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400718 in calc_array (w=1920, h=1080) at stackflow.c:16
16              llBuf[i]=i*2;
(gdb) p i
$1 = 0  //访问llBuf[0]就出现了段错误

uos@uos-PC:~$ sudo trace-bpfcc -t 'r::acct_stack_growth "retval=%d",retval' -p 3180
请输入密码:
验证成功
TIME     PID    TID    COMM         FUNC          -
16.04381 3180   3180   stackflow    acct_stack_growth retval=-12 //返回的错误码为-12
```

结论：

llBuf是一个局部变量，访问llBuf[0]就出现了段错误，并不是常识中的访问更大的地址空间出现了段错误，这个疑问需要搞清楚，同时访问错误码-12也需要搞清楚。

统信软件 UNIONTECH

10.3.3)acct_stack_growth retval返回-12理解

已知acct_stack_growth代码如下，到底哪种情况，通过BPF+GDB调试确认

```
int acct_stack_growth(struct vm_area_struct *vma,
                      unsigned long size, unsigned long grow)
{
    struct mm_struct *mm = vma->vm_mm;
    unsigned long new_start;

    /* address space limit tests */
    if (!may_expand_vm(mm, vma->vm_flags, grow))
        return -ENOMEM;

    /* Stack limit test */
    if (size > rlimit(RLIMIT_STACK))
        return -ENOMEM;

    /* mlock limit tests */
    if (vma->vm_flags & VM_LOCKED) {
        unsigned long locked;
        unsigned long limit;
        locked = mm->locked_vm + grow;
        limit = rlimit(RLIMIT_MEMLOCK);
        limit >>= PAGE_SHIFT;
        if (locked > limit && !capable(CAP_IPC_LOCK))
            return -ENOMEM;
    }

    /* Check to ensure the stack will not grow into a hugetlb-only region */
    new_start = (vma->vm_flags & VM_GROWSUP) ? vma->vm_start :
            vma->vm_end - size;
    if (is_hugepage_only_range(vma->vm_mm, new_start, size))
        return -EFAULT;
```

1)扩展vma失败，返回内存不足

2)超过栈空间大小限制，返回内存不足

3)申请的内存为LOCKED类型，超过了LOCKED大小限制，返回内存不足

统信软件 UNIONTECH

10.3.4)acct_stack_growth retval返回-12理解

a)确认栈空间申请大小

sudo trace-bpfcc -t 'acct_stack_growth(struct vm_area_struct *vma,unsigned long size)
"siz=%d",size' -p 3308

TIME PID TID COMM FUNC -

6.249033 3308 3308 stackflow acct_stack_growth siz=16646144

16646144B=15.875M

b)rlimit(RLIMIT_STACK))返回的值是多少呢?

它返回的是tsk->signal->rlim[limit].rlim_cur

首先写一个kprobe代码获取current进程的地址

然后基于gdb打印tsk->signal->rlim[limit].rlim_cur的值

统信软件
UNIONTECH

10.3.5)acct_stack_growth retval返回-12理解

c)获取current进程地址的代码如下

```
#!/usr/bin/bpftrace
BEGIN
{       printf("begin trace nvme with kprobe, pid=%d\n", $1);}
kprobe:acct_stack_growth
{
        if (pid == $1) {
                printf("name=%s curtask=0x%llx\n", comm, curtask);//打印current的地址
        }
}
END
{       printf("finished\n"); }
```

已知kprobe返回的地址为0xffff8026f36e4240

d)GDB获取进程栈大小

```
sudo gdb ./vmlinux-187 /proc/kcore
(gdb) p ((struct task_struct*)0xffff8026f36e4240)->signal->rlim[3].rlim_cur
$1 = 8388608  //gdb得到的栈限制大小为8388608=8M
```

统信软件
UNIONTECH

10.3.6)acct_stack_growth调用理解

基于BPF获取到的调用acct_stack_growth的堆栈如下：

sudo trace-bpfcc -tK 'acct_stack_growth(struct vm_area_struct *vma,unsigned long size) "siz=%d",size' -p 3503

```
acct_stack_growth+0x0 [kernel]
expand_stack+0xc [kernel]
__do_page_fault+0x84 [kernel]
do_page_fault+0x144 [kernel]
do_translation_fault+0x58 [kernel]
do_mem_abort+0x3c [kernel]
el0_da+0x20 [kernel]
```

可以看到栈内存空间也是基于vma申请内存地址、基于缺页原理申请内存.

继续确认为什么给llbuf[0]赋值时就出来了段错误异常,因为常识中都是从低地址写，一边写一边申请内存，当内存不够时，申请内存失败也就出来了内存改写，所以常识中应该是如访问llBuf[8M]附近的地址时出现了错误。

统信软件
UNIONTECH

10.3.7)acct_stack_growth栈扩展理解

expand_downwards调用了acct_stack_growth, 该函数内部有如下代码:

```
if (address < vma->vm_start) { //这个判断条件满足意味着超过了目前的栈空间限制
  size = vma->vm_end - address;//计算需要扩展的栈空间大小
}
```

size的大小为：

```
 p 0x1000000000000 - 0xffffff02be70
$5 = 16597392
```

结论：

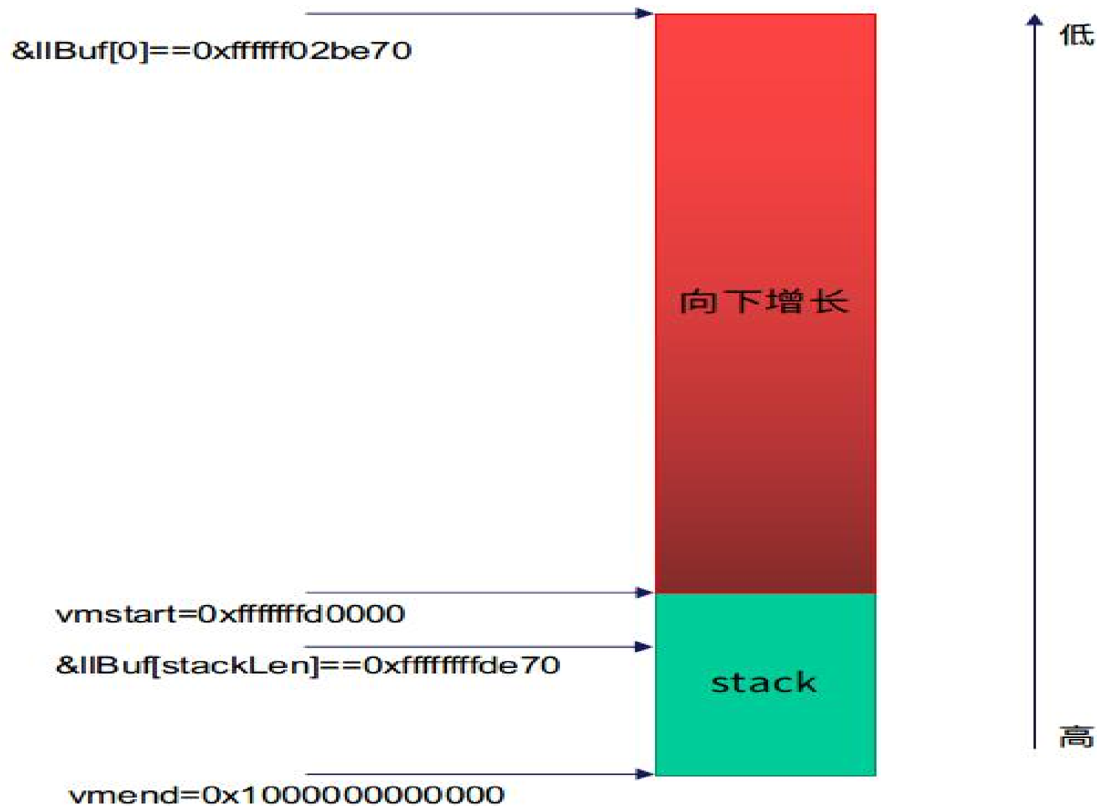
访问&llBuf[0]=0xffffff02be70, 但由于栈空间的内存申请是向下增长，那么导致的结果就是访问低地址的内存时反而要申请大内存地址。

注意这儿是申请大内存地址，还不是物理页内存，物理页内存基于缺页原理申请

非栈空间申请内存时只要管理地址空间是否够用就可以了，但栈的空间申请多了一个栈大小的限制判断，默认为8M，现在扩展的大小超过了8M， 这个是返回内存失败的真正原因。

统信软件
UNIONTECH

10.3.8)栈扩展原理图



统信软件 UNIONTECH

11. 预编译调试

统信软件
UNIONTECH

11)引言

一般开源代码都会使用大量的宏，特别是内核，到处基于配置 和宏来定义代码，本文描述了如何基于预编译的方案来快速理解这些宏相关的变量、代码。

本节以虚拟地址为例子来描述如何借助预编译来快速理解虚拟地址相关的代码和原理。

统信软件
UNIONTECH

11.1)简单例子

```
#define VA_BITS            48
#define PAGE_OFFSET    (UL(0xffffffffffffffff) - \
    (UL(1) << (VA_BITS - 1)) + 1)

int main(int argc, char** argv)  {
      long pageoffset = PAGE_OFFSET;// 直接将宏赋值给变量，可以通过预编译展开来看结果
      return 0;
}
```

统信软件
UNIONTECH

11.1.1)简单例子预编译

gcc -E usemacro.c -o usemacro.E

usemacro.E中展开的代码如下：

```
int main(int argc, char** argv) {
    long pageoffset = (UL(0xffffffffffffffff) - (UL(1) << (48 - 1)) + 1);//宏已经展开了
 return 0;
}
```

预编译没有将最终的值给出来, 可以使用gdb来计算

```
p /x  (0xffffffffffffffffUL - ((1UL) << (48 - 1)) + 1)
0xffff800000000000
```

总结：-E表示要预编译，预编译做的主要工作就是将所有的宏展开

统信软件
UNIONTECH

11.2)虚拟地址

虚拟地址分为用户态虚拟地址和内核态虚拟地址，用户态虚拟地址全部为页表地址，内核态虚拟地址又分为线性地址和页表虚拟地址。

内核有对这三种虚拟地址的计算，这些虚拟地址如何有效划分的呢？看到地址后知道是哪种类型的地址吗？本文通过理解内核相关代码来解答这个疑问。

统信软件
UNIONTECH

11.2.1)用户态页表虚拟地址

理解用户态虚拟地址的关键是找到申请虚拟地址的地方,mmap是最好的分析场景

基于以下代码来分析：

```
#include <sys/mman.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int main(int argc, char** argv) {
    int fd=open("aa.mmap", O_CREAT);
    void *myaddr= mmap(0, 1024*1024*1024, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0);
    printf("fd=%d myaddr=0x%lx\n", fd, myaddr);
    getchar();
    close(fd);
    return 0;
}
```

统信软件
UNIONTECH

11.2.2)用户态虚拟地址结果查看

运行上面的代码，看到的地址如下：

fd=3 myaddr=0xfffe8d910000

myaddr=0xff fe 8d 91 00 00

用户态虚拟地址为48位，地址为48位是因为内核编译时的这个配置：

CONFIG_ARM64_VA_BITS=48

统信软件
UNIONTECH

11.2.3)mmap的调用流程

想要知道虚拟地址的划分，最有效的方法就是看虚拟地址的生成，如这儿的mmap返回的虚拟地址的生成。由于本节的重点讲解如何借用预编译分析问题，因此这儿不详细分析mmap的内部流程。

对于arm64系统，从系统调用函数出发到最终的内存申请，整个调用流程如下：

arch/arm64/kernel/sys.c::mmap --> ksys_mmap_pgoff --> vm_mmap_pgoff --> do_mmap_pgoff --> do_mmap --> get_unmapped_area --> thp_get_unmapped_area --> arch_get_unmapped_area_topdown --> vm_unmapped_area --> unmapped_area_topdown

unmapped_area_topdown完成虚拟地址的申请，不过申请的虚拟地址从哪儿开始受mm->mmap_base的影响，这个设置代码就在arch_get_unmmaped_area_topdown函数。

统信软件
UNIONTECH

11.2.4)mmap_base地址的使用

这段代码就在arch_get_unmmaped_area_topdown函数，虚拟地址默认为下向增长方式，即这儿的mm->mmap_base是最大的，后面的地址都比它小。

```
289      info.flags = VM_UNMAPPED_AREA_TOPDOWN;
290      info.length = len;
291      info.low_limit = max(PAGE_SIZE, mmap_min_addr);
292      if (is_exagear_compat_task())
293          info.high_limit = mm->context.exagear_mmap_base;
294      else
295          info.high_limit = mm->mmap_base;
296      info.align_mask = 0;
297      addr = vm_unmapped_area(&info);
298
```

1）使用此地址作为虚拟地址的基地址，后面的地址都比这小

2)开始申请虚拟地址

统信软件 UNIONTECH

11.2.5)mmap_base地址的生成

上一节知道了虚拟地址从mm->mmap_base这个基地址开始申请，那么这个值是如何计算出来的是我们最关心的事情了。

该地址的计算在mmap_base函数中完成，execve启动进程时计算出来。调用mmap_base的堆栈如下：

```
arch_pick_mmap_layout+0x0 [kernel]
load_elf_binary+0x368 [kernel]
search_binary_handler+0xbc [kernel]
__do_execve_file.isra.12+0x4ec [kernel]
do_execve+0x2c [kernel]
__arm64_sys_execve+0x28 [kernel]
el0_svc_common+0x90 [kernel]
el0_svc_handler+0x9c [kernel]
el0_svc+0x8 [kernel]
```

统信软件
UNIONTECH

11.2.6)mmap_base地址的生成

mmap_base代码如下：

```
static unsigned long mmap_base(unsigned long rnd, struct rlimit *rlim_stack)
{
	unsigned long gap = rlim_stack->rlim_cur;
	unsigned long pad = stack_guard_gap;

	/* Account for stack randomization if necessary */
	if (current->flags & PF_RANDOMIZE)
		pad += (STACK_RND_MASK << PAGE_SHIFT);

	/* Values close to RLIM_INFINITY can overflow. */
	if (gap + pad > gap)
		gap += pad;

	if (gap < MIN_GAP)
		gap = MIN_GAP;
	else if (gap > MAX_GAP)
		gap = MAX_GAP;

	return PAGE_ALIGN(STACK_TOP - gap - rnd);
}
```

需要看看STACK_TOP及PAGE_ALIGN这些宏的定义

统信软件 UNIONTECH

11.2.7)STACK_TOP宏

由于STACK_TOP这个宏比较简单，因此直接查看宏的定义就可以了

```
#define STACK_TOP      (test_thread_flag(TIF_32BIT) ? \
            AARCH32_VECTORS_BASE : STACK_TOP_MAX)
```

64位时肯定取后者，所以使用STACK_TOP_MAX

```
#define STACK_TOP_MAX      TASK_SIZE_64
```

```
#define TASK_SIZE_64      (UL(1) << VA_BITS)
#define VA_BITS       (CONFIG_ARM64_VA_BITS)
```

CONFIG_ARM64_VA_BITS=48，所以STACK_TOP的的值为0x1000000000000

统信软件 UNIONTECH

11.2.8)确认mmap_base的最大值

```
static unsigned long mmap_base(unsigned long rnd, struct rlimit *rlim_stack)
{
	unsigned long gap = rlim_stack->rlim_cur;
	unsigned long pad = stack_guard_gap;

	/* Account for stack randomization if necessary */
	if (current->flags & PF_RANDOMIZE)
		pad += (STACK_RND_MASK << PAGE_SHIFT);

	/* Values close to RLIM_INFINITY can overflow. */
	if (gap + pad > gap)
		gap += pad;

	if (gap < MIN_GAP)
		gap = MIN_GAP;
	else if (gap > MAX_GAP)
		gap = MAX_GAP;

	return PAGE_ALIGN(STACK_TOP - gap - rnd);
}
```

1)gap默认为堆栈大小8M, 但它小于MIN_GAP=128M, 所以gap被修改为128M

2)rnd为一个随机值，就是因为这个随机值，最终需要调用PAGE_ALIGN来进行对齐。

mmap_base的最大地址由STACK_TOP和gap决定。48位地址情况下，

如果随机值为0, 那么这个地址就是0xfffff8000000，所以用户态虚拟地址的最大值是不会超过0x00 00 ff ff f8 00 00 00的

再看前面mmap返回的虚拟地址0xfffe8d910000， 它确实要比0xfffff8000000小

11.3)内核页表页表虚拟地址

内核虚拟地址分为线性虚拟地址和页表虚拟地址。

为了性能最优，内核以线性虚拟地址为主，但为了满足一些特殊的功能，如内核有时要申请不连续的物理内存，此时就要页表原理来映射物理内存了，这也是用户态可以做到虚拟地址连续但物理地址可以不连续的本质原因。

从我目前对内核的理解来看，内核申请页表虚拟地址的函数还只发现了vmalloc, 我个人觉得只有这个函数的原因是上面的解释，页表虚拟地址是为了实现这个功能而用的，不然就用更好的线性地址了。

统信软件
UNIONTECH

11.3.1)vmalloc

Vmalloc --> __vmalloc --> __vmalloc_node --> __vmalloc_node_range

最终的内存申请在这儿, 可以看到通过VMALLOC_START和VMALLOC_END就可以知道内核页表虚拟地址的区间了

```
static void *__vmalloc_node(unsigned long size, unsigned long align,
                gfp_t gfp_mask, pgprot_t prot,
                int node, const void *caller)
{
    return __vmalloc_node_range(size, align, VMALLOC_START, VMALLOC_END,
                gfp_mask, prot, 0, node, caller);
}
```

vmalloc的虚拟地址从VMALLOC_START开始，止于VMALLOC_END

VMALLOC_START和VMALLOC_END涉及到很多相关的宏，为了快速准确知道它的值， 这儿采用预编译的方式来得知。

统信软件
UNIONTECH

11.3.2)预编译

1)首先将内核编译的打印放开

```
export KBUILD_VERBOSE=1
```

2)然后对想要编译的代码增加一个错误，如：

```
static void *__vmalloc_node(unsigned long size, unsigned long align,
            gfp_t gfp_mask, pgprot_t prot,
            int node, const void *caller)
{
#error "1111"
   return __vmalloc_node_range(size, align, VMALLOC_START, VMALLOC_END,
            gfp_mask, prot, 0, node, caller);
}
```

3)多进程快速编译让其出现错误，这个对于修改共公头文件情况很有效

```
make bindeb-pkg -j100
```

4)出现错误，停止下来后用一个进程编译

```
make bindeb-pkg -j1
```

统信软件
UNIONTECH

11.3.2)预编译

5)将编译命令copy出来修改

```
gcc -Wp,-MD,mm/.vmalloc.o.d  -nostdinc -isystem /usr/lib/gcc/aarch64-linux-gnu/8/include -I./arch/arm64/include -I./arch/arm64/include/generated  -I./include -I./arch/arm64/include/uapi -I./arch/arm64/include/generated/uapi -I./include/uapi -I./include/generated/uapi -include ./include/linux/kconfig.h -include ./include/linux/compiler_types.h -D__KERNEL__ -mlittle-endian -Wall -Wundef -Wstrict-prototypes -Wno-trigraphs -fno-strict-aliasing -fno-common -fshort-wchar -Werror-implicit-function-declaration -Wno-format-security -std=gnu89 -fno-PIE -mgeneral-regs-only -DCONFIG_AS_LSE=1 -fno-asynchronous-unwind-tables -Wno-psabi -mabi=lp64 -fno-delete-null-pointer-checks -Wno-frame-address -Wno-format-truncation -Wno-format-overflow -Wno-int-in-bool-context -O2 --param=allow-store-data-races=0 -Wframe-larger-than=2048 -fno-stack-protector -Wno-unused-but-set-variable -Wno-unused-const-variable -fno-omit-frame-pointer -fno-optimize-sibling-calls -fno-var-tracking-assignments -g -fno-inline-functions-called-once -Wdeclaration-after-statement -Wno-pointer-sign -Wno-stringop-truncation -fno-strict-overflow -fno-merge-all-constants -fmerge-constants -fno-stack-check -fconserve-stack -Werror=implicit-int -Werror=strict-prototypes -Werror=date-time -Werror=incompatible-pointer-types -Werror=designated-init -fmacro-prefix-map=./= -Wno-packed-not-aligned    -DKBUILD_BASENAME='"vmalloc"' -DKBUILD_MODNAME='"vmalloc"' -E -o mm/.tmp_vmalloc.E mm/vmalloc.c
```

-c变成了-E,表示预编译，输出从.o变成了.E

6)将前面人为添加的错误还原

```
static void *__vmalloc_node(unsigned long size, unsigned long align,
            gfp_t gfp_mask, pgprot_t prot,
            int node, const void *caller)
{
#error "1111"    还原
    return __vmalloc_node_range(size, align, VMALLOC_START, VMALLOC_END,
            gfp_mask, prot, 0, node, caller);
}
```

统信软件 UNIONTECH

11.3.3)预编译

7)打开预编译文件，根据关键字搜索

如，这儿关键__vmalloc_node_range这个关键字将会看到如下信息

```
static void *__vmalloc_node(unsigned long size, unsigned long align,
        gfp_t gfp_mask, pgprot_t prot,
        int node, const void *caller)
{
 return __vmalloc_node_range(size, align, (((((((0xffffffffffffffffUL))) - ((((1UL))) << (48)) + 1) + (0)) + (0x08000000))),
)) << ((48) - 1)) + 1) - (1UL << ((16 - 3) * (4 - (4 - 3)) + 3)) - ((((1UL))) << ((48) - 16 - 1 + 6)) - 0x00010000),
    gfp_mask, prot, 0, node, caller);
}
```

VMALLOC_START的宏展开结果

统信软件
UNIONTECH

11.3.4)从预编译中取出结果分析

VMALLOC_START=((((((0xffffffffffffffffUL))) - ((((1UL))) << (48)) + 1) + (0)) + (0x08000000))

=0xffff000000000000 + 0x08000000

=0xffff000008000000

所以内核页表地址为64位，其中第48位为0

#define VMALLOC_END (PAGE_OFFSET - PUD_SIZE - VMEMMAP_SIZE - SZ_64K)

VMALLOC_END=(((((0xffffffffffffffffUL))) - ((((1UL))) << ((48) - 1)) + 1) - (1UL << ((16 - 3) * (4 - (4 - 3)) + 3)) - ((((1UL))) << ((48) - 16 - 1 + 6)) - 0x00010000)

=0xffff7bdfffff0000

VMALLOC_END的最大地址0xffff7bdfffff0000不可能超过PAGE_OFFSET=0xffff800000000000

结论：

当有效地址为48位时，高16位（第48～63位）统一为标志位。

只能全为0或者全为1，全为0为用户空间，全为1为内核空间。

因为用户空间全为0，所以说打印的时候被忽略了，内核空间全为1，就会被打印出来。

用户态地址向下增长，不可能出现地址相交的情况。

进一步看内核态线性虚拟地址

统信软件
UNIONTECH

11.3.5)内核态线性虚拟地址

可以直接查看物理地址转虚拟地址的宏，也通过上面的预编译方法来快速得知，如下：

```
#define __phys_to_virt(x)   ((unsigned long)((x) - PHYS_OFFSET) | PAGE_OFFSET)
PHYS_OFFSET =0
PAGE_OFFSET = (UL(0xffffffffffffffff) - (UL(1) << (VA_BITS - 1)) + 1) =0xffff800000000000
```

内核态的线性虚拟地址从0xffff800000000000开始向上增长

结论：

内核态线性虚拟地址第48位为1, 内核态页表虚拟地址的第48位为0， 且线性地址从更大的值向上增长，也不存在地址相交的情况

统信软件 UNIONTECH

11.3.6)总结

A)48位地址情况下，0xffff000000000000是区分内核态虚拟和用户态虚拟地址的分界线
0xffff000000000000开始表示是内核态的虚拟地址，向上增长； 0x0000ffff80000000用户态虚拟址，向下增长。

B)48位地址情况下，0xffff800000000000是内核态页表虚拟地址和线性虚拟地址的分界线，比0xffff800000000000大的都是线性虚拟地址，比它小的都是页表虚拟地址

C)内核态页表虚拟地址主要是vmalloc使用，具体的开始、结束地址是【0xffff000008000000，0xffff7bdfffff0000】

D) 【0xffff7e0000000000， 0xffff800000000000UL】是内核struct page本身

E) 内核及c库喜欢用宏，预编译是分析宏相关代码的有效方法之一

统信软件 UNIONTECH

11.3.6)总结

F)关于虚拟地址的划分

4.19内核的Documentation/arm64/memory.txt一文中有如下描述,但这个描述和实际上的实现有所出入，从前面的分析可以知道，应该减去一个gap值128M, 所以最大地址为0xfffff8000000.

假设用bfp获取被调试进程的task地址，然后基于gdb查看,发现mmpa_base为0xfffff8000000

(gdb) p /x ((struct task_struct*)0xffff80279f6330c0)->mm->mmap_base

$4 = 0xfffff8000000

AArch64 Linux memory layout with 64KB pages + 3 levels:

Start	End	Size	Use
0000000000000000	0000ffffffffffff	256TB	user
ffff000000000000	ffffffffffffffff	256TB	kernel

对于用户态地址，内核还需要减去一个gap值128M，所以这个地址最大为0xfffff8000000

统信软件 UNIONTECH

11.3.5)总结

G)5.10文档对虚拟地址的划分描述更加详细，不过用户态空间的最高地址的描述没有更新：

```
AArch64 Linux memory layout with 4KB pages + 4 levels (48-bit)::

  Start                 End                     Size            Use
  -----------------------------------------------------------------------
  0000000000000000      0000ffffffffffff         256TB          user
  ffff000000000000      ffff7fffffffffff         128TB          kernel logical memory r
 [ffff600000000000      ffff7fffffffffff]         32TB          [kasan shadow region]
  ffff800000000000      ffff800007ffffff         128MB          bpf jit region
  ffff800008000000      ffff80000fffffff         128MB          modules
  ffff800010000000      fffffbffefffffff         124TB          vmalloc
  fffffbfff0000000      fffffbfffdffffff         224MB          fixed mappings (top dow
  fffffbfffe000000      fffffbfffe7fffff           8MB          [guard region]
  fffffbfffe800000      fffffbffff7fffff          16MB          PCI I/O space
  fffffbffff800000      fffffbffffffffff           8MB          [guard region]
  fffffc0000000000      fffffdffffffffff           2TB          vmemmap
  fffffe0000000000      ffffffffffffffff           2TB          [guard region]


AArch64 Linux memory layout with 64KB pages + 3 levels (52-bit with HW support)::

  Start                 End                     Size            Use
  -----------------------------------------------------------------------
  0000000000000000      000fffffffffffff           4PB          user
  fff0000000000000      ffff7fffffffffff          ~4PB          kernel logical memory r
 [fffd800000000000      ffff7fffffffffff]        512TB          [kasan shadow region]
  ffff800000000000      ffff800007ffffff         128MB          bpf jit region
  ffff800008000000      ffff80000fffffff         128MB          modules
  ffff800010000000      fffffbffefffffff         124TB          vmalloc
  fffffbfff0000000      fffffbfffdffffff         224MB          fixed mappings (top dow
  fffffbfffe000000      fffffbfffe7fffff           8MB          [guard region]
  fffffbfffe800000      fffffbffff7fffff          16MB          PCI I/O space
  fffffbffff800000      fffffbffffffffff           8MB          [guard region]
  fffffc0000000000      fffffdffffffffff          ~4TB          vmemmap
```

从内核目前的实现来看，这个不正确，因为还需要减一个gap，这个值是128M

统信软件 UNIONTECH

12. 页保护原理调试

统信软件
UNIONTECH

12.1)引言

内核基于分页原理管理内存，常见页大小有4K, 8K, 16K, 64K. 规划好了最小内存页,这样就方便为每个页配置一个数据结构来管理它了，这个数据结构就是struct page, 大小为64B, 有了数据结构也就方便给页增加各种控制信息了，如PG_dirty、PG_reclaim,PG_head等标志。

strcut page只是在内核态使用，暴露到用户态的只有虚拟地址，因此一切管理从虚拟地址出来，具体来说有两方面。

一个是对虚拟地址本身进行管理，每一段映射出来的虚拟地址，内核都会建立一个strcut vma来管理它，虚拟地址的开始地址、结束地址、访问属性都会记录在vma中。mmap函数返回一段虚拟地址时内核态就已经创建好了管理它的vma, 当读写此内存时，由于没有映射物理地址，mmu上报异常，操作系统处理缺页异常时，首先会根据地址找到对应的vma, vma提供了地址范围、页访问属性、所属的struct mm_struct内存上下文信息等，这些是操作系统能够进行权限校验、页表访问的关键。

统信软件 UNIONTECH

12.1)引言

vma只有操作系统知道，mmu不知道，从mmu也需要有一个能够访问页地址和页属性的地方，当页地址没有触发缺页异常，当访问权限错误触发访问异常。这些都依赖pte。

如果虚拟地址没有对应的物理地址，此时基于缺页原理来申请内存，此时就会涉及到pgd, pud,pmd,pte等页表，由于虚拟地址和物理页是一一对应起来的，每个物理页都有对应的pte来管理它，由于pte保存物理不需要用完64位，如64K页大小时低16位是没有使用，48位地址时高16位没有使用，此时页的属性都可以保存到pte项中。如果调用mprotect函数设置地址的访问属性为只读的，那么PAGE_READONLY就会被保存到pte中，PAGE_READONLY根据VM_READ映射而来， VM_READ等于用户态能使用的PROT_READ.

总结起来就是操作系统基于vma就知道虚拟地址的访问权限，mmu需要基于pte知道虚拟地址的访问权好限(pud, pmd类型的页基于pud,pmd来知道，因为没有pte)

本文先讲解页权限是如何设置到pte的，然后讲解如何基于虚拟地址保护原理写一个内存改写检查工具。

统信软件
UNIONTECH

12.2)修改地址属性简单例子

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <string.h>
int main(int argc, char** argv) {
    long len=1024*128;
    printf("pid=%d\n",getpid());
    getchar();
    char *myaddr= (char*)mmap(0, len, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, 0, 0);
    strcpy(myaddr, "0123456789012345");
    mprotect(myaddr, len, PROT_READ);
    myaddr[0]='b';
    getchar();
    return 0;
}
```

统信软件
UNIONTECH

12.3)例子解释

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <string.h>
int main(int argc, char** argv) {
    long len=1024*128;
    printf("pid=%d\n",getpid());          // 1)打印pid, 方便bpf跟踪调试
    getchar();
    char *myaddr= (char*)mmap(0, len, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, 0, 0);   // 2)申请匿名内存，具有读写权限，大写为128K
    strcpy(myaddr, "0123456789012345");   // 3)此时能写，因为虚拟地址具有写权限
    mprotect(myaddr, len, PROT_READ);     // 4）修改为只有读权限
    myaddr[0]='b';                        // 5）此时再去写就会出现段错误了
    getchar();
    return 0;
}
```

统信软件 UNIONTECH

12.3.1)mprotect理解

mprotect就是修改传入地址段的访问属性,修改多少由传入的长度决定。

这儿主要查看mprotect最后是如何将虚拟地址的属性设置到pte上去的。

主要调用流程如下:

mprotect --> mprotect_fixup

mprotect_fixup中将属性设置到pte上的关键代码如下

```
vma->vm_flags = newflags;
dirty_accountable = vma_wants_writenotify(vma, vma->vm_page_prot);
vma_set_page_prot(vma);

change_protection(vma, start, end, vma->vm_page_prot,
          dirty_accountable, 0);
```

1)PROT_READ转成了VM_READ,并保存到了vm_flags

2)vma->vm_flags设置到了vma->vm_page_prot

3)将vma->vm_page_prot写到pte

统信软件 UNIONTECH

12.3.2)vma_set_page_prot理解

vma_set_page_prot完成了vma属性格式到页属性格式的转换,只有转换成了页属性格式才能保存到pte中。

调用流程为vma_set_page_prot --> vm_pgprot_modify --> vm_get_page_prot -->vm_get_page_prot.

vm_get_page_prot的关键代码如下：

```
pgprot_t vm_get_page_prot(unsigned long vm_flags)
{
    pgprot_t ret = __pgprot(pgprot_val(protection_map[vm_flags &
                (VM_READ|VM_WRITE|VM_EXEC|VM_SHARED)]) |
            pgprot_val(arch_vm_get_page_prot(vm_flags)));

    return arch_filter_pgprot(ret);
}
EXPORT_SYMBOL(vm_get_page_prot);
```

1)属性格式的转换就是查表，protection_map中保存的是页属性格式，VM_READ等是vma的属性格式

统信软件 UNIONTECH

12.3.3)protection_map理解

protection_map的定义如下：

```
pgprot_t protection_map[16] __ro_after_init = {
100    __P000, __P001, __P010, __P011, __P100, __P101, __P110, __P111,
101    __S000, __S001, __S010, __S011, __S100, __S101, __S110, __S111
102 };
```

以VM_READ为例，VM_READ=1, 所以查看protection_map[1]对应的__P001，对于arm64:

```
#define __P001  PAGE_READONLY
#define PAGE_READONLY       __pgprot(_PAGE_DEFAULT | PTE_USER | PTE_RDONLY | PTE_NG | PTE_PXN | PTE_UXN)
#define PTE_USER       (_AT(pteval_t, 1) << 6)     /* AP[1] */
#define PTE_RDONLY     (_AT(pteval_t, 1) << 7)
#define PTE_AF         (_AT(pteval_t, 1) << 10)    /* Access Flag */
#define PTE_NG         (_AT(pteval_t, 1) << 11)    /* nG */
#define PTE_PXN        (_AT(pteval_t, 1) << 53)    /* Privileged XN */
#define PTE_UXN        (_AT(pteval_t, 1) << 54)    /* User XN */
```

只读属于是一系列属性的缓合，最好理解的是PTE_RDONLY,它表示只读，所有这些属性占用位都是64位的低16位和高16位，因此能保存到pte项中，这也是需要将vma属性转换成pte属性的关键，只有转换才能保存到pte中。

统信软件 UNIONTECH

12.3.4)属性格式的保存

vma_set_page_prot将属于保存到了vma->vm_page_prot中，然后传给了change_protection,所以从change_protection出发，一直到最终的设置设置的属性都是vma->vm_page_prot

```
static inline void vma_set_page_prot(struct vm_area_struct *vma)
{
    vma->vm_page_prot = vm_get_page_prot(vma->vm_flags);
}
```

1）转换的属性保存到了vm_page_prot中

```
vma->vm_flags = newflags;
dirty_accountable = vma_wants_writenotify(vma, vma->vm_page_prot);
vma_set_page_prot(vma);

change_protection(vma, start, end, vma->vm_page_prot,
        dirty_accountable, 0);
```

2调用change_protection时通过vma->vm_page_prot传入

统信软件
UNIONTECH

12.3.5)change_protection理解

protection_map的定义如下：

change_protection --> change_protection_range -->change_p4d_range --> change_pud_range -->change_pmd_range --> change_pte_range

change_pte_range中有如下关键代码

根据addr,pmd找到pte

pte = pte_offset_map_lock(vma->vm_mm, pmd, addr, &ptl);

将pte内部的值先获取出来，然后清0

ptent = ptep_modify_prot_start(mm, addr, pte);

将新的属性值newprot新增进来

ptent = pte_modify(ptent, newprot);

总结：

mprotect除了将虚拟地址访问属性设置vma外，还通过调用change_protection将页访问属性设置到了pte

统信软件 UNIONTECH

12.3.6)修改属性前后对比

mmap(0, len, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, 0, 0)调用之后的情况:

```
00400000-00410000 r-xp 00000000 08:04 743445678                          /home/uos/zhoupeng/train/mmap
00410000-00420000 rw-p 00000000 08:04 743445678                          /home/uos/zhoupeng/train/mmap
00420000-00450000 rw-p 00000000 00:00 0                                  [heap]
ffff7e00000-fffff7f60000 r-xp 00000000 08:03 201338164                   /usr/lib/aarch64-linux-gnu/libc-2.28.so
fffff7f60000-fffff7f70000 rw-p 00150000 08:03 201338164                  /usr/lib/aarch64-linux-gnu/libc-2.28.so
fffff7f70000-fffff7fa0000 rw-p 00000000 00:00 0
fffff7fa0000-fffff7fb0000 r--p 00000000 00:00 0                          [vvar]
fffff7fb0000-fffff7fc0000 r-xp 00000000 00:00 0                          [vdso]
fffff7fc0000-fffff7fe0000 r-xp 00000000 08:03 201338160                  /usr/lib/aarch64-linux-gnu/ld-2.28.so
fffff7fe0000-fffff7ff0000 r--p 00010000 08:03 201338160                  /usr/lib/aarch64-linux-gnu/ld-2.28.so
fffff7ff0000-fffff8000000 rw-p 00020000 08:03 201338160                  /usr/lib/aarch64-linux-gnu/ld-2.28.so
fffffffd0000-1000000000000 rw-p 00000000 00:00 0                         [stack]
```

此时的地址还是rw, 即可读可写

mprotect(myaddr, len, PROT_READ);

```
fffff7e00000-fffff7f60000 r-xp 00000000 08:03 201338164                  /usr/lib/aarch64-linux-gnu/libc-2.28.so
fffff7f60000-fffff7f70000 rw-p 00150000 08:03 201338164                  /usr/lib/aarch64-linux-gnu/libc-2.28.so
fffff7f70000-fffff7f80000 rw-p 00000000 00:00 0
fffff7f80000-fffff7fa0000 r--p 00000000 00:00 0
fffff7fa0000-fffff7fb0000 r--p 00000000 00:00 0                          [vvar]
fffff7fb0000-fffff7fc0000 r-xp 00000000 00:00 0                          [vdso]
fffff7fc0000-fffff7fe0000 r-xp 00000000 08:03 201338160                  /usr/lib/aarch64-linux-gnu/ld-2.28.so
fffff7fe0000-fffff7ff0000 r--p 00010000 08:03 201338160                  /usr/lib/aarch64-linux-gnu/ld-2.28.so
fffff7ff0000-fffff8000000 rw-p 00020000 08:03 201338160                  /usr/lib/aarch64-linux-gnu/ld-2.28.so
fffffffd0000-1000000000000 rw-p 00000000 00:00 0                         [stack]
```

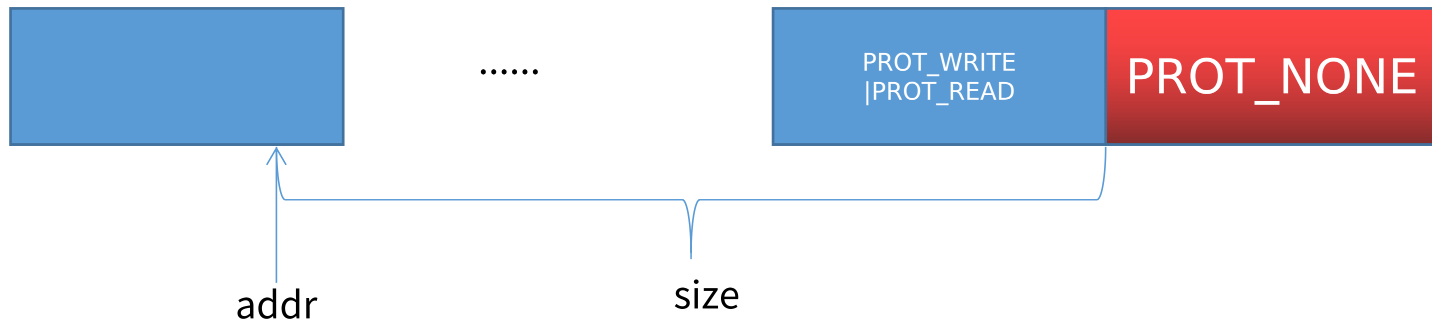
2)myaddr=0xfffff7f80000, 这个地址已经变成只读的了

信软件
ONTECH

12.4)基于页保护原理写内存改写工具

内存改写检查工具实现原理

下图展示的是申请内存大小大于1页的情况，如本例中的65539B大小



统信软件 UNIONTECH

12.4.1)原理解释

内存改写检查工具实现原理

统信软件
UNIONTECH

12.4.2)简单的内存改写检测代码

```
#include <stdlib.h>
#include <sys/mman.h>
#define PAGE_SHIFT 16
#define PAGE_SIZE (1<<PAGE_SHIFT)
void *malloc(size_t size) {
   int pages = (size + PAGE_SIZE -1)/PAGE_SIZE+ 1;

   int len = pages << PAGE_SHIFT;

   char *addr = (char*)mmap(0, len, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, 0,0);

   mprotect(addr+len - PAGE_SIZE, PAGE_SIZE, PROT_NONE);

   char *retaddr= addr+(PAGE_SIZE-(size%PAGE_SIZE));
   return retaddr;
}int main(int argc, char** argv) {
   char* mybuf = (char*)malloc(65539);
   mybuf[65538]=0;

   mybuf[65539]=0;
   return 0;
}
```

统信软件
UNIONTECH

12.4.2)代码解释

```
#include <stdlib.h>
#include <sys/mman.h>
#define PAGE_SHIFT 16
#define PAGE_SIZE (1<<PAGE_SHIFT)
void *malloc(size_t size) {
  int pages = (size + PAGE_SIZE -1)/PAGE_SIZE+ 1;
  int len = pages << PAGE_SHIFT;
  char *addr = (char*)mmap(0, len, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, 0,0);
  mprotect(addr+len - PAGE_SIZE, PAGE_SIZE, PROT_NONE);
  char *retaddr= addr+(PAGE_SIZE-(size%PAGE_SIZE));
  return retaddr;
}int main(int argc, char** argv) {
  char* mybuf = (char*)malloc(65539);
  mybuf[65538]=0;
  mybuf[65539]=0;
  return 0;
}
```

1）假设一页大小为64K

2)根据大小求出需要申请的物理页数

3)根据页数求出申请的内存大小

4)基于mmap申请内存大小

5)将最后一页设置为不可访问

6)根据实际大小访问地址，保证一个字节都不多

7)这个地址还是可写的

8）越界一个字节，因为页保护原理出现段错误

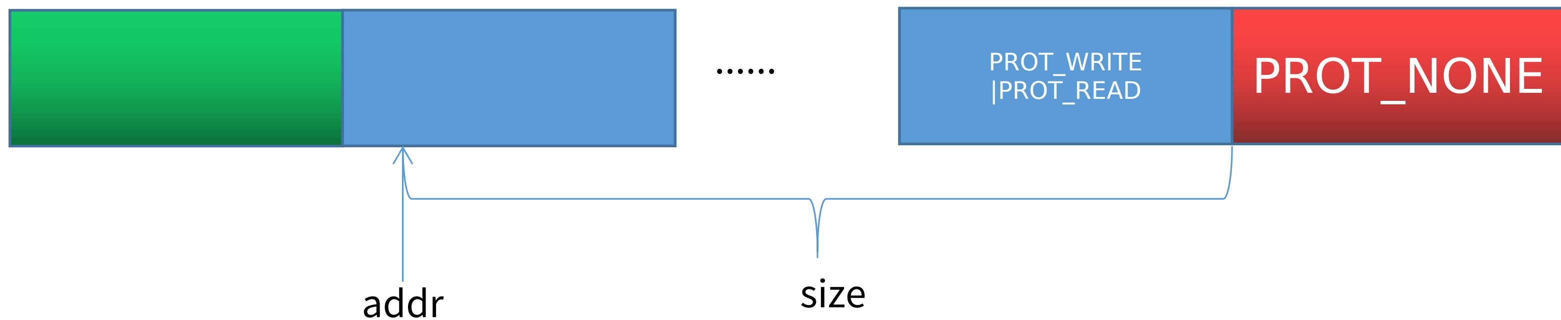
统信软件 UNIONTECH

12.5)总结

1)基于页保护原理的内存改写检查工具能做到第一现场，做到改写时就触发段错误异常，再配合堆栈打印功能，能够准确的找到改写现场。

2)本示例比较简单，申请出来的内存地址需要自己管理起来，如动态申请大数组理的内存，确保正确释放。

3)也可以做成将内存的管理信息保存到这儿的多余内存中，假设管理信息需要32B信息，那么当偏移的头中多余32B就保存在里面，当小于32B时，就再多申请一页，保存在下面绿色的页面里面。也可以总是多申请这一样，统一保存在里面，因为这儿可以避免保护的内容被改写。



4）这儿的内存申请全部基于mmap申请，需要修改进程的最大mmap次数，防止因为mmap申请次数受限导致申请内存失败

统信软件 UNIONTECH

13. valgrind

统信软件
UNIONTECH

13)引言

valgrind是我用过的工具中最强大的内存泄露、内存改写检查工具，对于内存泄露它的强大之处主要有两点。

第一点：将内存泄露分类，如分为必然的内存泄露、可能的内存泄露及可达的但没有释放的内存泄露；

第二点：可以将泄露的大小、堆栈打印出来。

对于内存改写，同样可以打印出现内存访问异常及改写的堆栈。

当然valgrind提供的功能远远不止这样，还有强大的cpu缓存统计工具cachegrind,堆内存使用分析工具，本节主要讲解最基础但也最常用的内存泄露和内存改写检查功能。

13.1)内存泄露

一般我们自己写内存泄露检查工具都是要求退出，因为退出了进程还没有释放，那肯定就是内存泄露了。

不过对于valgrind它可以支持两种场景，一种是进程退出来的场景，这个非常适合做单元测试、集成测试的进程，因为这些测试进程通常都会退出。

第二种场景是进程不退出场景，这个比较适合真实的业务进程，特别是后台服务进程。

第二种场景也能检查是因为valgrind能精准地分析内存泄露的类型，同时还有堆栈打印功能，借助这两方面的信息去分析代码通常都能将内存泄露全部检查出来。

统信软件
UNIONTECH

13.2)能退出的内存泄露用例

```
#include <stdlib.h>
int main(int argc, char** argv) {
    char *psz = malloc(32);//申请内存，但是没有释放，看看valgrind是如何报告的
    return 0;
}
```

```
valgrind --leak-check=full ./memleak
```

统信软件 UNIONTECH

13.2.2)内存泄露执行

```
uos@uos-PC:~/zhoupeng/train/method13$ valgrind --leak-check=full ./memleak
==6230== Memcheck, a memory error detector
==6230== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6230== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==6230== Command: ./memleak
==6230==

==6230==
==6230== HEAP SUMMARY:
==6230==     in use at exit: 32 bytes in 1 blocks
==6230==   total heap usage: 1 allocs, 0 frees, 32 bytes allocated
==6230==
==6230== 32 bytes in 1 blocks are definitely lost in loss record 1 of 1
==6230==    at 0x4863DD4: malloc (in /usr/lib/aarch64-linux-gnu/valgrind/vgpreload_memcheck-arm64-linux.so)
==6230==    by 0x4005AB: main (memleak.c:3)
==6230==
==6230== LEAK SUMMARY:
==6230==    definitely lost: 32 bytes in 1 blocks
==6230==    indirectly lost: 0 bytes in 0 blocks
==6230==      possibly lost: 0 bytes in 0 blocks
==6230==    still reachable: 0 bytes in 0 blocks
==6230==         suppressed: 0 bytes in 0 blocks
==6230==
==6230== For counts of detected and suppressed errors, rerun with: -v
==6230== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

1)告诉我们进程退出时有一次内存泄露。大小为32字节。1 allocs知道是一次申请

2)definitely明确是泄露

3）泄露的堆栈也详细地打印出来了，malloc函数已经被valgrind接管

4)可以看到内存泄露的分类,如何理解可以看帮助文档，也可以自己写用例来确认，如definitely就是没有指针指向它了。reachable就是用全局指针指向但没有释放

13.3)不能退出的内存泄露用例

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int memleak1(int size) {
    char *psz = malloc(size);
}
int memleak2(int size) {
    char *psz = malloc(size);
}
int memleak3(int size) {
    char *psz = malloc(size);
}
char *g_psz1=0;
char *g_psz2=0;
int memleak4(int size) {
    g_psz1 = malloc(size);
}
int memleak5(int size) {
    g_psz2 = malloc(size);
}
```

```
int main(int argc, char** argv) {
    char *psz = malloc(32);
    printf("begin to alloc memory pid=%d\n", getpid());
    memleak1(1111);

    printf("press any key to alloc memory:2456\n");
    getchar();      memleak2(2456);

    printf("press any key to alloc memory:356\n");
    getchar();      memleak3(356);

    printf("press any key to alloc memory:45678\n");
    getchar();      memleak4(45678);

    printf("press any key to alloc memory:560000\n");
    getchar();      memleak5(560000);

    printf("finish alloc memory\n");      getchar();
    return 0;
}
```

统信软件
UNIONTECH

13.3.2)不能退出的内存泄露用例

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int memleak1(int size) {
    char *psz = malloc(size);
}
int memleak2(int size) {
    char *psz = malloc(size);
}
int memleak3(int size) {
    char *psz = malloc(size);
}
char *g_psz1=0;
char *g_psz2=0;
int memleak4(int size) {
    g_psz1 = malloc(size);
}
int memleak5(int size) {
    g_psz2 = malloc(size);
}
```

1)写多个函数是为了让valgrind的报告丰富一些

2)再新增两个函数，但申请的内存赋给两个全局变量

```c
int main(int argc, char** argv) {
    char *psz = malloc(32);
    printf("begin to alloc memory pid=%d\n", getpid());
    memleak1(1111);

    printf("press any key to alloc memory:2456\n");
    getchar();      memleak2(2456);
    printf("press any key to alloc memory:356\n");
    getchar();      memleak3(356);

    printf("press any key to alloc memory:45678\n");
    getchar();      memleak4(45678);

    printf("press any key to alloc memory:560000\n");
    getchar();      memleak5(560000);

    printf("finish alloc memory\n");     getchar();
    return 0;
}
```

3)赋个局部变量

4)申请不同大小，申请一次等待一次,方便查看

统信 UNION

13.3.3)用例运行

由于进程不退出，此时需要借助gdb来动态打印堆栈, valgrind巧妙地集成了自己修改后的gdb, 这儿称之为vgdb。

此时的调试分为服务+客户模式， 用例子来说明更好理解一些

服务端:

valgrind --leak-check=full --vgdb=yes --vgdb-error=0 ./memleakloop

客户端:

gdb ./memleakloop

target remote|/usr/bin/vgdb --pid=5760

gdb是我们自己安将的gdb

/usr/bin/vgdb是valgrind内部提供的gdb

统信软件
UNIONTECH

13.3.4)服务端启动

```
uos@uos-PC:~/zhoupeng/train/method13$ valgrind --leak-check=full --vgdb=yes --vgdb-error=0 ./memleakloop
==6339== Memcheck, a memory error detector
==6339== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6339== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==6339== Command: ./memleakloop
==6339==
==6339== (action at startup) vgdb me ...
==6339==
==6339== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==6339==   /path/to/gdb ./memleakloop
==6339== and then give GDB the following command
==6339==   target remote | /usr/lib/aarch64-linux-gnu/valgrind/../../bin/vgdb --pid=6339
==6339== --pid is optional if only one valgrind process is running
==6339==
```

1)表示服务端已经启动了gdb监听接口，等待客户端连接调试

2)详细地告诉我们了客户端如何去调试

UNIONTECH

13.3.5)客户端启动

gdb ./memleakloop

target remote|/usr/bin/vgdb --pid=6339

链接成功后按c命令继续，此时服务端的memleakloop进程才会开始运行

当在服务端按任一键申请这几个内存后：

begin to alloc memory pid=6360

press any key to alloc memory:2456

press any key to alloc memory:356

在客户端按ctrl+c看看此时的内存泄露情况

输入如下命令查看：

monitor leak_check reachable any

统信软件
UNIONTECH

13.3.6)客户端动态打印内存泄露

```
(gdb) monitor leak_check reachable any
==6360== 32 bytes in 1 blocks are still reachable in loss record 1 of 6
==6360==    at 0x4863DD4: malloc (in /usr/lib/aarch64-linux-gnu/valgrind/vgpreload_memcheck-arm64-linux.so)
==6360==    by 0x4007A7: main (memleakloop.c:23)
==6360==
==6360== 356 bytes in 1 blocks are definitely lost in loss record 2 of 6
==6360==    at 0x4863DD4: malloc (in /usr/lib/aarch64-linux-gnu/valgrind/vgpreload_memcheck-arm64-linux.so)
==6360==    by 0x40071F: memleak3 (memleakloop.c:12)
==6360==    by 0x4007FB: main (memleakloop.c:34)
==6360==
==6360== 1,024 bytes in 1 blocks are still reachable in loss record 3 of 6
==6360==    at 0x4863DD4: malloc (in /usr/lib/aarch64-linux-gnu/valgrind/vgpreload_memcheck-arm64-linux.so)
==6360==    by 0x4911D4B: _IO_file_doallocate (filedoalloc.c:101)
==6360==    by 0x4920B03: _IO_doallocbuf (genops.c:347)
==6360==    by 0x491FE67: _IO_file_overflow@@GLIBC_2.17 (fileops.c:752)
==6360==    by 0x491EFD3: _IO_new_file_xsputn (fileops.c:1251)
==6360==    by 0x491EFD3: _IO_file_xsputn@@GLIBC_2.17 (fileops.c:1204)
==6360==    by 0x48F67E7: vfprintf (vfprintf.c:1323)
==6360==    by 0x48FDD43: printf (printf.c:33)
==6360==    by 0x4007C3: main (memleakloop.c:25)
==6360==
==6360== 1,024 bytes in 1 blocks are still reachable in loss record 4 of 6
==6360==    at 0x4863DD4: malloc (in /usr/lib/aarch64-linux-gnu/valgrind/vgpreload_memcheck-arm64-linux.so)
==6360==    by 0x4911D4B: _IO_file_doallocate (filedoalloc.c:101)
==6360==    by 0x4920B03: _IO_doallocbuf (genops.c:347)
==6360==    by 0x491FC13: _IO_file_underflow@@GLIBC_2.17 (fileops.c:493)
==6360==    by 0x4920BA7: _IO_default_uflow (genops.c:362)
==6360==    by 0x4007DB: main (memleakloop.c:29)
```

1)main函数中的内存申请

2)memleak3申请内存

3)打印内部申请的缓冲区

4) 打印申请

13.3.7)客户端动态打印内存泄露

```
==6360==
==6360== 1,111 bytes in 1 blocks are definitely lost in loss record 5 of 6
==6360==    at 0x4863DD4: malloc (in /usr/lib/aarch64-linux-gnu/valgrind/vgpreload_memcheck-arm64-linux.so)
==6360==    by 0x4006D7: memleak1 (memleakloop.c:6)
==6360==    by 0x4007CB: main (memleakloop.c:26)
==6360==
==6360== 2,456 bytes in 1 blocks are definitely lost in loss record 6 of 6
==6360==    at 0x4863DD4: malloc (in /usr/lib/aarch64-linux-gnu/valgrind/vgpreload_memcheck-arm64-linux.so)
==6360==    by 0x4006FB: memleak2 (memleakloop.c:9)
==6360==    by 0x4007E3: main (memleakloop.c:30)
==6360==
==6360== LEAK SUMMARY:
==6360==    definitely lost: 3,923 bytes in 3 blocks
==6360==    indirectly lost: 0 bytes in 0 blocks
==6360==      possibly lost: 0 bytes in 0 blocks
==6360==    still reachable: 2,080 bytes in 3 blocks
==6360==         suppressed: 0 bytes in 0 blocks
==6360==
```

5)memleak1申请内存

6)memleak2申请内存

7)内存泄露进行了详细分类

13.4)内存改写

只要是经过malloc申请的内存, 全部都能进行内存改写的监控，由于valgrind对内存的申请进行了接管，不再走c库的申请。

请看下面的简单例子

```
#include <stdlib.h>
int main(int argc, char** argv)  {
    char *psz = malloc(32);//申请32B内存
    psz[32]='b';//一字节内存改写
    return 0;
}
```

```
valgrind --tool=memcheck --leak-check=full   ./memcorrupt
```

统信软件
UNIONTECH

13.4.2)检查内存改写

```
uos@uos-PC:~/zhoupeng/train/method13$ valgrind --tool=memcheck --leak-check=full   ./memcorrupt
==9799== Memcheck, a memory error detector
==9799== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==9799== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==9799== Command: ./memcorrupt
==9799==
==9799== Invalid write of size 1
==9799==    at 0x4005BC: main (memcorrupt.c:4)
==9799==  Address 0x4a30060 is 0 bytes after a block of size 32 alloc'd
==9799==    at 0x4863DD4: malloc (in /usr/lib/aarch64-linux-gnu/valgrind/vgpreload_memcheck-arm64-linux.so)
==9799==    by 0x4005AB: main (memcorrupt.c:3)
==9799==
==9799==
==9799== HEAP SUMMARY:
==9799==     in use at exit: 32 bytes in 1 blocks
==9799==   total heap usage: 1 allocs, 0 frees, 32 bytes allocated
==9799==
==9799== 32 bytes in 1 blocks are definitely lost in loss record 1 of 1
==9799==    at 0x4863DD4: malloc (in /usr/lib/aarch64-linux-gnu/valgrind/vgpreload_memcheck-arm64-linux.so)
==9799==    by 0x4005AB: main (memcorrupt.c:3)
==9799==
==9799== LEAK SUMMARY:
==9799==    definitely lost: 32 bytes in 1 blocks
==9799==    indirectly lost: 0 bytes in 0 blocks
==9799==      possibly lost: 0 bytes in 0 blocks
==9799==    still reachable: 0 bytes in 0 blocks
==9799==         suppressed: 0 bytes in 0 blocks
==9799==
```

1)运行命令

2)提示改写一字节

3)内存改写堆栈

4)本身也是内存泄露

UNIONTECH

13.5)理解valgrind的内存改写机制

valgrind很大，还没有时间去看它的实现机制，但是可以简单通过BPF+GDB+VGDB+Valgrind来确认。

如确认是否基于页保护原理来监控内存改写。

```
#include <stdlib.h>
int main(int argc, char** argv) {
    char *psz = malloc(32);
    psz[32]='b';
    char *psz2 = malloc(1024);  //如果基于页保护原理，那么此时还会申请内存
    psz2[100]='c';
    return 0;
}
```

统信软件
UNIONTECH

13.5.2)设计探测方法

控测方法如下:

a)Valgrind+VGDB启动带内存改写的进程, 它现在是一个服务端

b)GDB以客户端方式连接服务端

c)客户端连接上以后对main设置断点

d)单步跟踪，看看执行malloc时的物理页的申请情况

e)单步跟踪，看看写申请的内存时物理页的申请情况

f)继续单步跟踪，看看第二次申请的内存时物理页的申请情况

g)继续单步跟踪，看看第二次写内存时物理页的申请情况

e)c 继续运行

统信软件
UNIONTECH

13.5.3)具体操作步骤

执行步骤如下:

a)valgrind --tool=memcheck --leak-check=full --vgdb=yes --vgdb-error=0 ./memcorrupt

提示启动的进程的pid=9918

target remote | /usr/lib/aarch64-linux-gnu/valgrind/../../bin/vgdb --pid=9918

b)BFP跟踪9918进程的内存申请

sudo trace-bpfcc -tK '__alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order, int preferred_nid) "order=%d nid=%d", order, preferred_nid' -p 9918

c)客户端连接

gdb ./memcorrupt

target remote|/usr/bin/vgdb --pid=9648

d)b main设置断点

统信软件
UNIONTECH

13.5.4)测试结果解析

第一个malloc(32)触发了4个物理页内存的申请

182.2199 9918　9918　memcheck-arm64- __alloc_pages_nodemask order=0 nid=2

182.2199 9918　9918　memcheck-arm64- __alloc_pages_nodemask order=0 nid=2

182.2200 9918　9918　memcheck-arm64- __alloc_pages_nodemask order=0 nid=2

182.2200 9918　9918　memcheck-arm64- __alloc_pages_nodemask order=0 nid=2

psz[32]='b'没有触发物理页内存的申请，这说明valgrind在申请内存时就进行了管理，如填充了一些值，特别是真对一字节改写。

第二个char *psz2 = malloc(1024)也没有申请内存，写内存也是。

这说明valgrind没有基于页保护原理来监控内存的改写，但它又能做到改写时就报错，怎么做到的呢?

直接基于memcheck工具的代码分析肯定能找到原因。

统信软件 UNIONTECH

13.6)分析memcheck代码

首先根据memcheck工具的提示搜索代码

因为出现错误时有如下打印：

Invalid write of size 1

在memcheck下面grep -rns --include=*.c “Invalid” .|grep size

结果发现./memcheck/mc_errors.c文件下面的MC_(pp_Error)函数有如下代码：

```
case Err_Addr:
         emit( "Invalid %s of size %lu\n",
             extra->Err.Addr.isWrite ? "write" : "read",
             extra->Err.Addr.szB );
```

MC_(pp_Error)函数就是vgMemCheck_pp_Error, 现在告诉它的错误码是Err_Addr

基于Err_Addr搜索进一步发现MC_(record_address_error)函数触发

进一步搜索谁调用了MC_(record_address_error), 并通过增加打印发现是mc_STOREVn_slow调用了。

统信软件 UNIONTECH

13.6.2)分析mc_STOREVn_slow函数

mc_STOREVn_slow函数有如下关键代码：

```
1644     for (i = 0; i < szB; i++) {
1645        PROF_EVENT(MCPE_STOREVN_SLOW_LOOP);
1646        ai      = a + byte_offset_w(szB, bigendian, i);
1647        vbits8 = vbytes & 0xff;
1648        ok      = set_vbits8(ai, vbits8);
1649        if (!ok) n_addrs_bad++;
1650        vbytes >>= 8;
1651     }
1652     DEBUG_ZP("zhoupeng3 invoke5 record_zp_address_error n_addrs_bad=%d\n", n_addrs_bad);
1653     /* If an address error has happened, report it. */
1654     if (n_addrs_bad > 0)
1655        MC_(record_address_error)( VG_(get_running_tid)(), a, szB, True );
1656  }
1657
```

1)ok这个返回值决定了是否有错误

2)自己增加一条打印来确认

3)出现了错误，就准备调用该函数打印错误信息了

现在有两个需要理清：

1)谁调用了mc_STOREVn_slow

2)set_vbits8是如何认为有错误的

统信软件 UNIONTECH

13.6.3)设计代码调试

问题了方便跟踪是谁调用了mc_STOREVn_slow

为了方便调用设计如下调试代码：

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char** argv) {
  char *psz = malloc(31);
    //打印pid方便连接上去调试
    printf("press any key to contine pid=%d, psz=0x%lx psz[32]=0x%lx\n", getpid(), psz, &psz[32]);
    getchar();//启动起来后在这儿等待，方便GDB连接上去调试
    psz[32]='b';//就只调试这一条语句
    getchar();
    return 0;
}
```

统信软件
UNIONTECH

13.6.4)设计GDB调试

1)用valgrind启动待调试的进程, valgrind本身不能启动gdb，不然即使主动加载了memcheck-arm64-linux, 客户端也无法读取这个库的符号，提示没有权限

/home/uos/zhoupeng/valgrind/valgrind-3.14.0/coregrind/valgrind --tool=memcheck ./memcorrupt

2)gdb ./memcorrupt

attach pid

3)连接上去后主动让gdb加载memcheck-arm64-linux

symbol-file /usr/local/lib/valgrind/memcheck-arm64-linux

需要保证/usr/local/lib/valgrind有如下几个自己统计的文件

default.supp   vgpreload_core-arm64-linux.so

memcheck-arm64-linux vgpreload_memcheck-arm64-linux.so

4)设置断点

b memcheck/mc_main.c:1648

1648行的代码为这个：   ok   = set_vbits8(ai, vbits8);

统信软件
UNIONTECH

13.6.5)GDB调试

当执行psz[32]='b'这条语句时获取到的堆栈如下:

```
#0  get_secmap_for_reading (a=77791328) at mc_main.c:1648
#1  get_vabits2 (a=77791328) at mc_main.c:765
#2  set_vbits8 (vbits8=0 '\000', a=77791328) at mc_main.c:802
#3  mc_STOREVn_slow (a=77791328, nBits=<optimized out>, vbytes=0, bigendian=<optimized out>)
   at mc_main.c:1648
#4  0x0000001003a29ec4 in ?? ()
#5  0x00000000595f4b58 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
```

这儿最大的疑问就是0x0000001003a29ec4这个地址，它是谁呢？真的是这个地址调用了mc_STOREVn_slow函数吗?

查看运行memcorrupt的进程的maps信息

1003890000-10059a0000 rwxp 00000000 00:00 0

0x0000001003a29ec4在这个地址范畴内，它是一个带执行属性的地址段，这说明它可以是代码，反汇编看看它是什么。

统信软件 UNIONTECH

13.6.6)GDB中反汇编0x0000001003a29ec0

0x0000001003a29ec4是返回后的下一条地址，所以反汇编它前面的代码

```
(gdb) disas 0x0000001003a29eb0,0x0000001003a29ec8
Dump of assembler code from 0x1003a29eb0 to 0x1003a29ec8:
   0x0000001003a29eb0:  mov    x7, #0x0                    // #0
   0x0000001003a29eb4:  mov    x1, x7
   0x0000001003a29eb8:  mov    x9, #0x6490                 // #25744
   0x0000001003a29ebc:  movk   x9, #0x5801, lsl #16
   0x0000001003a29ec0:  blr    x9
=> 0x0000001003a29ec4:  mov    x7, #0x62                   // #98
```

果然是代码，x9就是要去执行的代码

x9 = (0x5801 <<16) | 0x6490= 0x58016490

```
x /x (0x5801 <<16) | 0x6490
0x58016490 <vgMemCheck_helperc_STOREV8>:      0xf25b681f
```

结论：

0x0000001003a29ec0去执行vgMemCheck_helperc_STOREV8,最终执行到了mc_STOREVn_slow .

那么0x0000001003a29ec0所在的完整代码是怎样的？又是谁在调用这样的代码呢？

统信软件 UNIONTECH

13.6.7)确认psz[32]='b'这行代码的执行

从前面的分析可以知道0x0000001003a29ec0是真实存在的代码，它不属于memcheck-arm64-linux这个工具内部的代码，但最终会调用到memcheck-arm64-linux里面的代码完成访问地址的校验，所以0x0000001003a29ec0比较像是虚拟机代码，它替换了真实运行的代码。

为了确认这个猜想的正确性，可以对main函数中的psz[32]='b'这行代码的汇编代码设置断点来验证。

gdb连接成功能后，主动加载memcorrupt的符号：

symbol-file ./memcorrupt

然后disas main函数，基于getchar的调用，很快可以确认下面这块红色代码完成了psz[32]='b'

```
0x00000000004006b4 <+64>:   bl     0x400550 <getchar@plt>
0x00000000004006b8 <+68>:   ldr    x0, [sp, #40]
0x00000000004006bc <+72>:   add    x0, x0, #0x20
0x00000000004006c0 <+76>:   mov    w1, #0x62                // #98
0x00000000004006c4 <+80>:   strb   w1, [x0]
0x00000000004006c8 <+84>:   bl     0x400550 <getchar@plt>
```

因此b *0x4006b8, 看看这个地址是否会被调用, 结果gdb无法在这个地址停下来

总结：

基于valgrind启动时，没有运行memcorrupt进程本身的代码，如这儿的main函数中的代码，而是运行的valgrind提供的代码。0x0000001003a29ec0所在的代码就是valgrind生成的代码。

统信软件 UNIONTECH

13.6.8)查找执行虚拟机的代码

由于已经确定是生成机制，猜测有一个线程在不断生成运行，这个线程很有可能就是主线程，由于gdb attach上去后看到的堆栈如下：

```
(gdb) bt
#0  vgModuleLocal_do_syscall_for_client_WRK ()    at m_syswrap/syscall-arm64-linux.S:112
#1  0x0000001002017360 in ?? ()
```

搜索调用vgModuleLocal_do_syscall_for_client_WRK函数的代码，这就是do_syscall_for_client

对do_syscall_for_client设置断点，继续运行，获取到的堆栈如下：

```
(gdb) bt
#0  do_syscall_for_client (syscall_mask=0x10033bfce0, tst=0x1002017350, syscallno=63) at m_syswrap/syswrap-main.c:1964
#1  vgPlain_client_syscall (tid=tid@entry=1, trc=trc@entry=73) at m_syswrap/syswrap-main.c:1964
#2  0x00000000580a2058 in handle_syscall (tid=tid@entry=1, trc=<optimized out>) at m_scheduler/scheduler.c:1176
#3  0x00000000580a3b48 in vgPlain_scheduler (tid=tid@entry=1) at m_scheduler/scheduler.c:1498
#4  0x00000000580ea8ec in thread_wrapper (tidW=1) at m_syswrap/syswrap-linux.c:103
#5  run_a_thread_NORETURN (tidW=1) at m_syswrap/syswrap-linux.c:156
#6  0x0000000000000000 in ?? ()
Backtrace stopped: previous frame identical to this frame (corrupt stack?)
```

总结：

scheduler.c比较像是虚拟机的主流程，thread_wrapper是主线程处理函数，vgPlain_scheduler是虚拟机处理函数，可以重点看看此函数内部的实现

统信软件 UNIONTECH

13.6.9)理解vgPlain_scheduler

vgPlain_scheduler函数中看到了如下调用，从函数名字来看它比较像是去做翻译执行

```
run_thread_for_a_while( &trc[0],
                        &dispatch_ctr,
                        tid, 0/*ignored*/, False );
```

run_thread_for_a_while的第1005行又找到了如下代码：

```
 SCHEDSETJMP( tid,  jumped,
   VG_(disp_run_translations)(
     two_words,
     (volatile void*)&tst->arch.vex,
     host_code_addr
   )
 );
```

gdb调试发现disp_run_translation展开后是vgPlain_disp_run_translations, 它是一个汇编代码。它很有可能就是跳转到虚拟机执行的代码。

统信软件
UNIONTECH

13.6.10)调试run_thread_for_a_while

在run_thread_for_a_while函数的1005行设置断点调试

理解这个代码后发现tst->arch.vec.guest_PC就是原汇编代码， 而host_code_addr就是虚拟机代码

```
(gdb) display /x tst->arch.vex.guest_PC
1: /x tst->arch.vex.guest_PC = 0x49722ac
(gdb) display /x host_code_addr
2: /x host_code_addr = 0x1003a294b0
```

当第7次执行到这儿时，发现guest_PC变成了0x4006b8, 它就是psz[32]=’b’这条语句的第一行代码。

```
1: /x tst->arch.vex.guest_PC = 0x4006b8
2: /x host_code_addr = 0x1003a29e18
```

总结：

0x1003a29e18这个开始地址就是0x4006b8代码块对应的虚拟机代码，有必要跟踪这个虚拟机代码做了些什么了，它什么能够将内存改写检查出来了。

得到原因之前基本可以猜测虚拟机代码肯定是在执行真正的赋值之前做了地址合法性的检查，如是否可读，是否可写，而这个是能够进行内存改写检查的真正原因。

统信软件 UNIONTECH

13.6.11)调试虚拟机代码0x1003a29e18

这儿已经是虚拟机的代码，它先调用vgMemCheck_helper_LOADV64le完成psz这个地址的检验

```
(gdb) disas 0x0000001003a29e18,0x0000001003a29e68
Dump of assembler code from 0x1003a29e18 to 0x1003a29e68:
   0x0000001003a29e18:  ldur    w9, [x21, #8]
   0x0000001003a29e1c:  subs    w9, w9, #0x1
   0x0000001003a29e20:  stur    w9, [x21, #8]
   0x0000001003a29e24:  b.pl    0x1003a29e30  // b.nfrst
   0x0000001003a29e28:  ldur    x9, [x21]
   0x0000001003a29e2c:  br      x9
   0x0000001003a29e30:  ldr     x7, [x21, #264]
   0x0000001003a29e34:  add     x28, x7, #0x28
   0x0000001003a29e38:  ldr     x7, [x21, #1192]
   0x0000001003a29e3c:  cmp     x7, #0x0
   0x0000001003a29e40:  b.eq    0x1003a29e50  // b.none
   0x0000001003a29e44:  mov     x9, #0x66c8                     // #26312
   0x0000001003a29e48:  movk    x9, #0x5801, lsl #16
   0x0000001003a29e4c:  blr     x9
   0x0000001003a29e50:  mov     x0, x28
   0x0000001003a29e54:  mov     x9, #0x5b18                     // #23320
   0x0000001003a29e58:  movk    x9, #0x5801, lsl #16
=> 0x0000001003a29e5c:  blr     x9
   0x0000001003a29e60:  mov     x7, x0
   0x0000001003a29e64:  neg     x6, x7
End of assembler dump.
(gdb) x $x9
0x58015b18 <vgMemCheck_helperc_LOADV64le>:      0xf25b7404
(gdb) p /x $x0
$172 = 0x1fff00dea8
(gdb) x $x0
0x1fff00dea8:   0x04a30040
(gdb)
```

1)x9是memcheck中提供的读内存代码
这个函数就是vgMemCheck_helperc_LOADV64le

3)这个地址保存到了0x1fff00dea8中，它应该是虚拟机的一部分

2）x0是要读的地址，就是psz的基地址

统信软件 UNIONTECH

13.6.12)校验基地址psz

#define VA_BITS16_NOACCESS 0x0000 #define VA_BITS16_UNDEFINED 0x5555 #define VA_BITS16_DEFINED 0xaaaa

0xaaaa代码此地址是有效的，0x5555代表读的地址无效。现在读的值为0xaaaa, 所以psz这个地址是有效的

```
0x58015b18 <vgMemCheck_helperc_LOADV64le>:    0xf25b7404
(gdb) p /x $x0
$172 = 0x1fff00dea8
(gdb) x $x0
0x1fff00dea8:   0x04a30040
(gdb) si
mc_LOADV64 (isBigEndian=0 '\000', a=137422233256) at mc_main.c:4930
4930        return mc_LOADV64(a, False);
(gdb) p /x a
$173 = 0x1fff00dea8
(gdb) si
0x0000000058015b1c      4930        return mc_LOADV64(a, False);
(gdb) si
get_secmap_for_reading_low (a=<optimized out>) at mc_main.c:4893
4893            vabits16 = sm->vabits16[sm_off16];
(gdb) n
mc_LOADV64 (isBigEndian=0 '\000', a=137422233256) at mc_main.c:4898
4898        if (LIKELY(vabits16 == VA_BITS16_DEFINED)) {
(gdb) p /x vabits16
$174 = 0xaaaa
(gdb) p sm_off16
No symbol "sm_of16" in cur...
(gdb) p sm_off16
$175 = 7125
(gdb) p 0x1fff00dea8
$176 = 137422233256
(gdb) p 0xdea8/8
$177 = 7125
(gdb)
```

1)vgMemCheck_helper_LOADV64le是去校验读的地址是否合法
如果读地址异常就可以返回异常了

4)vabits16=sm->vabits16[7125]=0xaaaa, 它是一个有效地址.
校验信息来至sm->vabits16，7125偏移来至虚拟地址a=0x1fff00dea8(虚拟机提供)

3）现在要读的地址0x1ffff00dea8, 0xdea8/8=7125, 所以读sm->vabits16[7125]它就是对psz这个地址的管理信息

```
zhoupeng3 invoke5 record_zp_address_error n_addrs
... (repeated)
press any key to contine pid=31789, psz=0x4a30040
a30060
```

```
258 #define VA_BITS32_UNDEFINED   0x55555555  // 01_01_01_01b x 4
259
260
261 #define SM_CHUNKS             16384     // Each SM covers 64k of memory.
262 #define SM_OFF(aaa)           (((aaa) & 0xffff) >> 2)
263 #define SM_OFF_16(aaa)        (((aaa) & 0xffff) >> 3)
264
265 // Paranoia:  it's critical for performance that the requested inlining
266 // occurs.  So try extra hard.
267 #define INLINE    inline __attribute__((always_inline))
```

1）根据地址获取此地址的检验偏移位

```
10030a0000-10032b0000 rwxp 00000000 00:00 0
10032b0000-10032c0000 ---p 00000000 00:00 0
10032c0000-10033c0000 rwxp 00000000 00:00 0
10033c0000-10033d0000 ---p 00000000 00:00 0
1003590000-10036e0000 rwxp 00000000 00:00 0
1003890000-10059a0000 rwxp 00000000 00:00 0
10064d0000-1006830000 rwxp 00000000 00:00 0
1fff000000-1fff010000 rw-p 00000000
fffe59660000-fffe59670000 r--p 0000
ffffcf410000-ffffcf440000 rw-p 00000000 00:00 0
uos@uos-PC:~/zhoupeng/train/method13$
```

13.6.13)校验写地址&psz[32]

psz[32]=’ b’ ，前面psz基地址已经校验，现在继续校验&psz[32]这个地址，现在是写场景，且只写一个字节，所以校验函数变成了vgMemCheck_helper_STOREV8

```
(gdb) disas 0x0000001003a29ea8,0x0000001003a29ed0
Dump of assembler code from 0x1003a29ea8 to 0x1003a29ed0:
   0x0000001003a29ea8:   blr     x9
   0x0000001003a29eac:   mov     x0, x28
   0x0000001003a29eb0:   mov     x7, #0x0                        // #0
   0x0000001003a29eb4:   mov     x1, x7
   0x0000001003a29eb8:   mov     x9, #0x6490                     // #25744
   0x0000001003a29ebc:   movk    x9, #0x5801, lsl #16
=> 0x0000001003a29ec0:   blr     x9
   0x0000001003a29ec4:   mov     x7, #0x62                       // #98
   0x0000001003a29ec8:   sturb   w7, [x28]
   0x0000001003a29ecc:   mov     x7, #0x0                        // #0
End of assembler dump.
(gdb) x $x9
0x58016490 <vgMemCheck_helperc_STOREV8>:        0xf25b681f
(gdb) p /x $x0
$182 = 0x4a30060
(gdb) p /x $x28
$183 = 0x4a30060
(gdb)
```

1)写之前先调用vgMemCheck_helper_STOREV8校验地址的合法性

2)如果成功就执行下面的赋值了，这儿x7就是'b'

x28就是要想的地址。

所以逻辑就是先校验，后执行

统信软件 UNIONTECH

13.6.14)校验写地址&psz[32]细节

vgMemCheck_helperc_STOREV8调用了mc_STOREVn_slow

mc_STOREVn_slow的最终校验在这儿:

```
Thread 1 "memcheck-arm64-" hit Breakpoint 8, get_secmap_for_reading (a=77791328)
    at mc_main.c:1648
1648            ok      = set_vbits8(ai, vbits8);
(gdb)
```

校验在这儿，ok代码可以访问

Set_vbits8调用了如下代码:

```
762   static INLINE
763   UChar get_vabits2 ( Addr a )
764   {
765     SecMap* sm      = get_secmap_for_reading(a);
766     UWord  sm_off  = SM_OFF(a);
767     UChar  vabits8  = sm->vabits8[sm_off];
768     return extract_vabits2_from_vabits8(a, vabits8);
769   }
```

sm保存的就是要访问的内存的值

(gdb) p /x sm->vabits8[16]@8

$194 = {0x55, 0x55, 0x55, 0x55, 0x55, 0x55, 0x55, 0x15}

Sm->vabits8[16]&8管理psz[0]~psz[31]的地址的有效访问

0x55 =01 01 01 01, 每个01表示一个地址的访问

0x15=00 01 01 01, 因为psz[31]不可访问。

sm_off=24, sm->vabits[24]=0x15, psz[31]在0x15中取出来的是0，所以psz[31]不可访问。

统信软件
UNIONTECH

13.6.15)校验完之后做什么

```
0x0000001003a29ec4 in ?? ()
(gdb) disas 0x0000001003a29ec0,0x0000001003a29ed0
Dump of assembler code from 0x1003a29ec0 to 0x1003a29ed0:
   0x0000001003a29ec0:   blr     x9
=> 0x0000001003a29ec4:   mov     x7, #0x62                    // #98
   0x0000001003a29ec8:   sturb   w7, [x28]
   0x0000001003a29ecc:   mov     x7, #0x0                     // #0
End of assembler dump.
(gdb) x $x28
0x4a30060:      0x00000000
(gdb) si
0x0000001003a29ec8 in ?? ()
(gdb) si
0x0000001003a29ecc in ?? ()
(gdb) x $x28
0x4a30060:      0x00000062
(gdb)
```

1)出现异常后继续执行后面的代码

2）修改之前是0,修改之后是0x62

上图中blr x9就是去执行vgMemCheck_helper_STOREV8， 即完成检验的过程，如果发现异常会有打印，但是即使发现了异常从blr x9返回后继续执行后面的mov x7, #0x62这样的代码。

可以看到这后面的代码完成了赋值的作用。

总结：

valgrind虽然提供了校验功能，即内存改写检查功能，但它不影响程序的执行，它只是在各个异常点做打印，帮我们发现问题。

统信软件
UNIONTECH

13.7)数组能检查出来吗?

设计如下代码

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
char g_psz[32]={0};
int main(int argc, char** argv) {
   //char *psz = malloc(31);
   printf("press any key to contine pid=%d, psz=0x%lx psz[32]=0x%lx\n", getpid(), g_psz,
&g_psz[32]);
   getchar();
   g_psz[32]='b'; //已经越界访问，但是没有报错
   getchar();
   return 0;
}
```

统信软件 UNIONTECH

13.7.2)数组不能检查原因分析

1）对vgMemCheck_helperc_STOREV8设置断点

因为只要是写就会调用此函数, 结果执行的关键几行代码在这儿:

```
vabits8 = sm->vabits8[sm_off];
  if (LIKELY(V_BITS8_DEFINED == vbits8)) {
    if (LIKELY(vabits8 == VA_BITS8_DEFINED)) {
      return; // defined on defined  }
```

2)计算sm_off

a=0x411068， sm_off = (0x411068 & 0xffff)/4=1050

```
(gdb) p /x sm->vabits8[1050]@4
$6 = {0xaa, 0xaa, 0xaa, 0xaa}
sm->vabits8[1050]=0xaa
#define VA_BITS8_DEFINED     0xaa
```

总结:

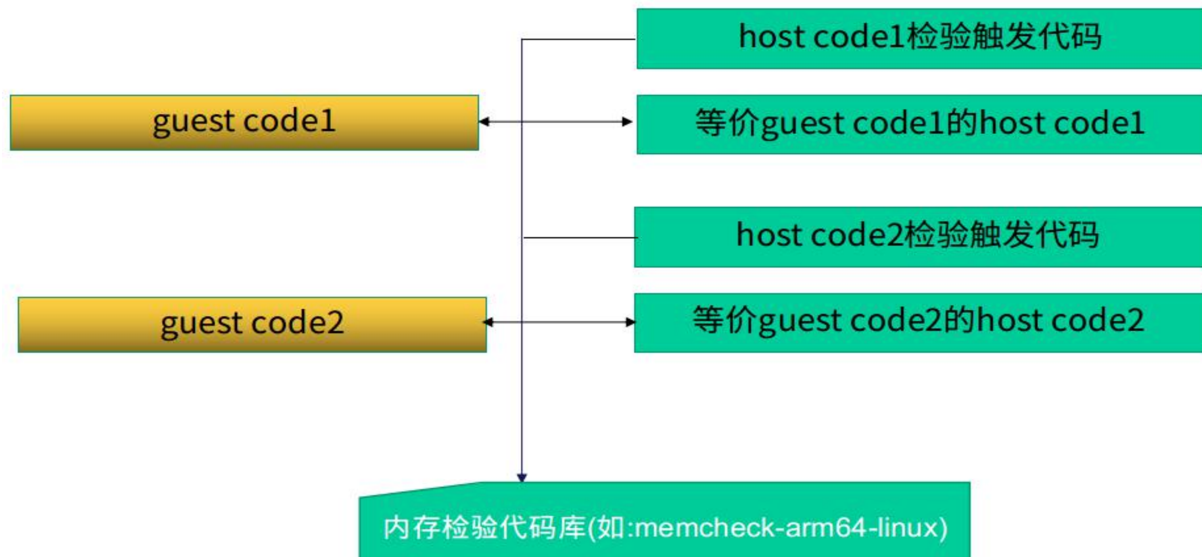
没有报错的原因是因为全局变量的地址空间没有标注，32B之外的地址空间虽然不是psz的，但也不是不可访问的，它有可能是其它全局变量的，如申请两个全局变量，会发现它们的地址是相邻的。不能像malloc申请的函数那样多申请一些空间，从而规划出不能访问的地址区域

统信软件
UNIONTECH

13.8)总结

1）valgrind的功能强大，内存改写、内存泄露的检查都具有堆栈打印功能，且内存改写检查基于第一现场打印堆栈

2）valgrind为了支持内存改写的第一现场监控功能，它实现了一个强大的代码生成器，将guest code生成host code. guest code就是被valgrind检查的代码，但它不会被运行，运行的全部都是host代码。代码生成器除了生成等价它的代码，还插入了一些带检验能的代码。这些检验代码是通用的，因此都直接实现成函数，供host代码调用。



统信软件
UNIONTECH

13.8)总结

3）被valgrind检查的代码在valgrind就是一个elf文件，这个elf文件被valgrind解析加载后，它是什么样的汇编代码valgrind也就都知道了，在运行之前它会生成一份对应的代码，生成的代码中插入了检验代码。所以需要把host代码想像成虚拟机要执行的代码，虚拟机则是schedule.c，主函数是vgPlain_scheduler. 所以可以把valgrind想像成valgrind虚拟机+valgrind代码生成器+公共检验函数库。

4）本节中没有去深入研究valgrind虚拟机的执行细节，也没有去看valgrind代码生成器是如何生成host code的，如果有一天需要实现一个类似的虚拟机就值得研究学习了。

5)调试读写是如何检查的，最好的跟踪点是run_thread_for_a_while函数中的准备去执行vgPlain_disp_run_translations这个函数的那一行代码，即m_scheduler/scheduler.c:1005.
假设我们知道了已经要运行的地址0x4006b8, 那么调试过程中只需要display /x txt-
>arch.vex.guest_PC
当tst->arch.vex.guest_PC = 0x4006b8==0x4006b8时，si汇编进去就可以调试整个过程了。
run_thread_for_a_while表示运行一会儿，到底运行多少呢？从我们的调式来看，我发现就是它以内存的读写的开始检查为分界线，执行完一个检查，txt->arch.vex.guest_PC这个地址就更新，没有必要连续。

统信软件
UNIONTECH

13.8)总结

6）valgrind使用帮助文档

https://valgrind.org/docs/manual/manual.html

7)本节主要理解了memcheck是如何做到基于第一现场完成内存改成检查的，从它的实现机制可以看到，它的用于管理的内存至少占申请的内存的1/4，因为第一个字节的内存都需要两位的bitmap内存去管理它。同时它的性能损失主要损失在检验上，基本上是每个地址的读写检查都有了校验，不慢是不可能的了。

检测流程参考下图

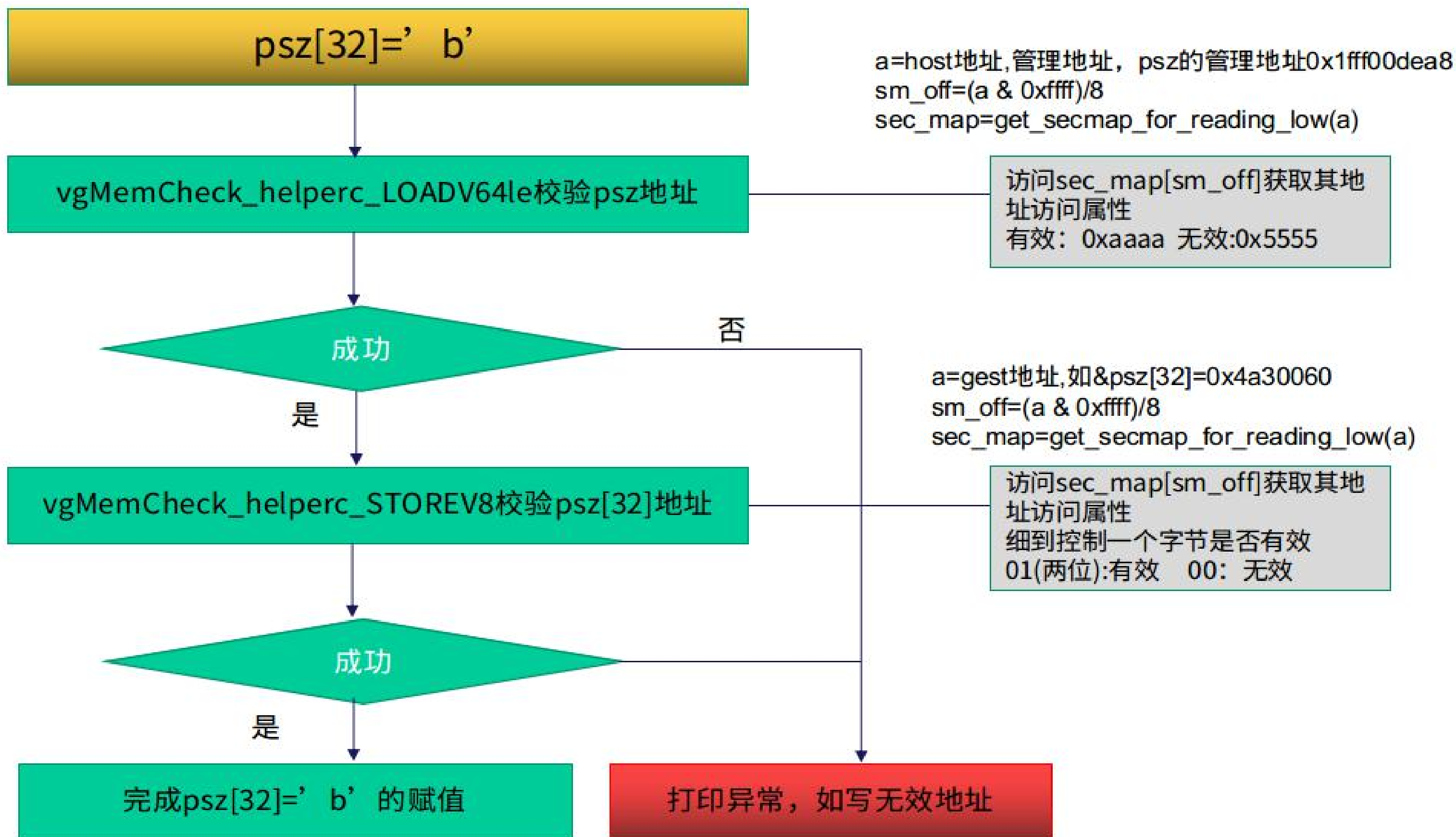
这个流程是host侧代码的流程，假设guest侧代码完成psz[32]=’b’赋值，那么host侧代码完成地址校验及赋值的功能。

如guest测运行首地址为0x4006b8， host测试运行首地址0x1003a29e18

这个代码就是0x1003a29e18~0x1003a29efc这段完整代码做的事情。

统信软件
UNIONTECH

13.8)总结



统信软件 UNIONTECH

14)总结

A)为了更容易让读者理解，本文中设计的例子都是对我们工作中复杂场景的关键问题的简单化模拟。大部分例子都可以copy下来直接运行，按照本文中的方法都可以操作演练。

B)文中提到的大部分方法都适合用来分析性能，对于稳定性问题，通常基于GDB就可以解决50%以上的问题了，如果再配上信号捕获、内存改写、内核泄露检查工具、日志等就能解决80%以上的问题了，当然这些都基于一个前提，那就是对相关问题背后机制和原理的快速理解。

C)授之以鱼、不如授之以渔，本文更多尝试从基础常识剖析、底层原理理解的角度来描述如何去分析一个未知问题，毕竟撑握分析问题的方法比直接得到结果更值得学习，实践出真知，希望这些隐藏在问题背后的方法和思路能给读者一些启迪。

D)GDB只能用来调试C/C++/ASM, 其它语言都有自己的调试器，但调试的思路是相通的，从我个人调试经验来看，学会GDB后，使用其它调试器调试相应代码时无师自通，唯一需要学习的是相应的执行命令。所以通过一种工具撑握调试能力很有意义。

统信软件
UNIONTECH

14.2)总结

E)由于时间、工作关系和能力限制，本文介绍的十二种调试方法不能覆盖我们工作所有遇到的问题. 如强大的valgrind主要用来检查内存泄露和内存改写，本文没有介绍；驱动出现异常时打印的堆栈如果只有地址没有符号如何使用addr2line还原代码；如何识别多线程导致的内存改写问题；如何快速识别死锁问题；如何在没有valgrind的情况下使用GDB识别一些内存改写问题；如何在多进程环境中识别出文件泄露导导某个功能异常问题等等。这些都是之前解决过的一些案例，从我个人的理解来看，这些都是将相应模块机制和代码细节理解后，基于本文中提高工具和分析问题的方法基本上都能解决。

G)本文中有一些机制的理解是和我的团队一块讨论后得出来的，如陈毅翀提供vmtouchX工进工具、叶中玉提供clear_page汇编代码、余晟锦提供内核互换调试思想，感谢原建业在写作过程的建义和初稿评审。讨论是工作中非常重要的一环，倾听问题、深入交流往往会有突破性发现，感谢我们团队背后的努力和支持！

The End

Thanks For Listening