《LINUX 平台高级调试与优化》试验指南

实验环境：

1， 推荐使用 Windows 操作系统，版本为 Windows 7 以上，必须为 64 位，如果使用 Mac 系统，某些步骤可能略有差异，请向讲师提问或者搜索网络寻求方案

2， 至少有 50GB 空闲磁盘空间

试验准备：

1， 运行 VirtualBox 安装程序，安装 VirtualBox 软件

2， 下载（地址细节请见邮件）或者复制虚拟机导出文件，如果为多卷压缩格式，请先解压缩，解压缩后应该是一个.ova 后缀的较大文件。

3， 启动 Virtual Box 的虚拟机管理器，点击“管理 > 导入虚拟机…”，选择实验材料中的 gedu.ova 文件，导入虚拟机，名为 GE64

4， 启动刚刚导入的虚拟机，以用户名 gedu 和密码 gedu（与用户名相同）登陆。需要使用 sudo 操作时，密码也是 gedu

5， 在 C 盘新建文件夹 c:\temp 用于与虚拟机交换文件，如果已经有不必重建

试验 1：使用 GDB 调试后台服务崩溃

试验目的：使用 GDB 调试 Linux 下后台服务(Daemon)崩溃原因，感受著名的头文件陷阱

试验环境：GE64 虚拟机

1. 启动和登陆 GE64 虚拟机，使用 Ctrl + Alt + T 热键打开一个终端窗口
2. 执行如下命令，感受段错误崩溃

```
$ cd labs/hdtrap
$ ./hdtrap
gedu@gedu-VirtualBox:~/labs/hdtrap$ ./hdtrap
running...
Segmentation fault (core dumped)
```

3. 执行命令 gdb ./hdtrap 开始 gdb 调试
4. 执行 b main 命令在主函数设置断点

```
Type "apropos word" to search for commands related to "word".
Reading symbols from ./hdtrap...done.
(gdb) b main
Breakpoint 1 at 0x80488e2: file hdtrap.c, line 100.
```

5. 执行 run Lushan 2018 命令运行下程，应该遇到断点而中断到 gdb

```
Breakpoint 1, main (argc=3, argv=0xffffd004) at hdtrap.c:100
100     {
```

6. 执行如下命令属性 gdb 的常用命令

```
(gdb) info args          # 观察参数
(gdb) info locals        # 观察局部变量
(gdb) p argv[0]@3        # 观察数组
```

```
(gdb) p argv[0]@3
$1 = {0xffffd1fa "/home/gedu/labs/hdtrap/hdtrap", 0xffffd218 "Lushan", 0xffffd21f "2018"}
```

(gdb) info shared　　# 观察进程里的共享模块

7. 执行 c 命令恢复下程执行，它很会会因为遇到段错误而中断

```
Program received signal SIGSEGV, Segmentation fault.
0x080489ca in calc_md5 (data=0x8048a60 "testing data-xxxxxxx", nLen=20, md5=0x0) at md5.c:7
7           md5[0] = A;
```

8. 执行 list 命令显示当前点附近的源代码
9. 执行 bt 命令观察当前线程的栈回溯（函数调用关系和参数）。注意观察 calc_md5 的第三个参数 md5 为空
10. 执行 print &md5 命令观察 md5 指针的地址

```
(gdb) print &fileid
$2 = (unsigned int **) 0xbffff2e4
```

11. 执行 frame 1 命令切换到父函数所在的栈帧，分析参数的来源
12. 执行 list 命令观察父函数（get_file_id）中调用子函数的代码
13. 执行 print sizeof(section)观察 section 结构体的大小
14. 执行 print sizeof(section.length)观察 length 字段的大小
15. 执行 pt section 命令观察 section 结构的定义
16. 执行 disassemble 命令观察当前栈帧对应的汇编指令，注意观察调用 calc_md5 函数前准备参数的过程。Call 指令上面的四条指令连续向栈上存放了 4×4=16 个字节的内容
17. 执行 info locals 观察局部变量的信息。执行 print fileid 观察 fileid 的值

```
(gdb) info locals
section = {data = 0x8048a00 "testing data-xxxxxxx", length = 20}
(gdb) print fileid
$5 = (unsigned int *) 0xbffff2f0
(gdb) print &fileid
$6 = (unsigned int **) 0xbffff2e4
```

18. 执行 frame 0 切换回发生段错误的 0 号栈帧，执行 info reg 观察寄存器的值。特别注意 ebp 寄存器的值，它代表的是栈帧基地址

```
esp            0xbffff2b0      0xbffff2b0
ebp            0xbffff2d8      0xbffff2d8
```

19. 执行 x/8x $ebp 显示当前栈帧以上的内容（比当前栈帧基地址大的数据），观察父函数传递过来的参数

```
(gdb) x/8x $ebp
0xbffff2a8:     0xbffff2d8      0x08048698      0x08048a00      0x00000014
0xbffff2b8:     0x00000000      0xbffff2f0      0xbffff308      0x08048a00
```

通常，EBP+8 的位置是第一个参数，其值为 0x080048a00，结合第 17 步，正是 section.data 的值，接下来的 0x00000014 即 20，正是 section.length 的值，接下来的 0x00000000 是 section.length 的高 32 位，再后面的 0xbffff2f0 是 fileid（参考 17 步的结果）。结合第 10 步观察到的 md5 参数的地址，正是 section.length 的高 32 位所在的位置。如此看来 calc_md5 函数索引参数时刚好错位了 4 个字节。父函数将 md5 参数放在了 0xbffff2bc 的位置，但是子函数使用的却是 0xbffff2b8 的位置。错位原因是，两个子函数对函数原型的理解不一样，子函数声明的原型是：

```
(gdb) pt calc_md5
type = int (char *, int, unsigned int *)
```

父函数中调用时，根据参数来推测函数原型，认为是：

int (char *, int64, unsigned int *)

而导致父函数误解的原因是没有申明子函数原型，也没有包含合适的头文件。

20. 执行 q 命令，选择 y，退出 GDB
21. 下面尝试使用 hdtrap 中的调试支持，在发生段错误后，自动启动 GDB。执行./hdtrap –jit 再次启动 hdtrap 程序，它打印出提示信息后，触发段错误，然后自动启动 GDB，如下图所示

```
[gedu@localhost hdtrap]$ ./hdtrap -jit
Stuntman for xsw by Raymond (rev1.0)
running...
jit debug handler registered
GNU gdb (GDB) Fedora (7.5.0.20120926-25.fc18)
Copyright (C) 2012 Free Software Foundation, Inc.
```

22. 执行 bt 命令可以看到启动 GDB 的过程，以及触发段错误的过程

```
#0  0xf772ec89 in __kernel_vsyscall ()
#1  0xf760e150 in nanosleep () from /lib32/libc.so.6
#2  0xf760e0a5 in sleep () from /lib32/libc.so.6
#3  0x08048819 in crash_handler (sig=11) at hdtrap.c:76
#4  <signal handler called>
#5  0x080489ca in calc_md5 (data=0x8048a60 "testing data-xxxxxxx", nLen=20, md5=0x0) at md5.c:7
#6  0x080486e6 in get_file_id (filename=0x8048afc "filea", fileid=0xffc1304c) at hdtrap.c:29
#7  0x0804899b in main (argc=2, argv=0xffc13114) at hdtrap.c:117
```

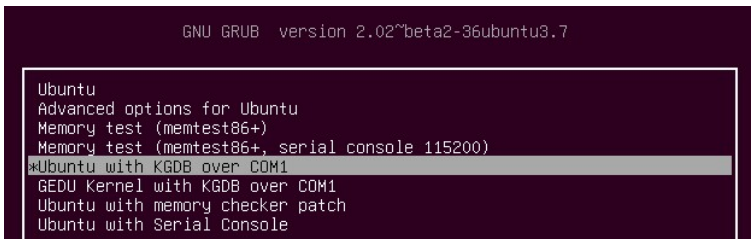
23. 执行 list 56,88 观察源代码中支持 JIT 调试的部分
24. 接下来可以重复上面的步骤，继续调试这个问题
25. 执行 stop 命令停止调试，执行 q 命令退出调试器，按 Ctrl + C 回到命令行

试验 2：使用 LINUX 双机内核调试探究句柄混论之谜

试验目的：使用 KGDB 观察应用层程序与驱动程序间通信的过程，分析数据混乱原因， 分时虚拟文件系统、Linux 驱动程序、sysfs、标准文件等设施

试验环境：在 VirtualBox 的管理控制台重复导入 gedu.ova 文件产生第二个虚拟机实例，命名为 GD64，以第二个实例作为调试主机，第一个实例（GE64）作为调试目标机，

1. 按下 SHIFT 键启动(如果已经启动则重启)GE64 虚拟机，调出 GRUB 菜单，按方向键，加亮 Ubuntu with KGDB over COM1 启动项，但是先不要按回车，让其等待



2. 在 VirtualBox 控制台中，选择 GD64 虚拟机，点击设置，然后在设置对话框的串口页面中，选择端口 1，选中“连接至现有通道或套接字”

3. 启动 GD64 虚拟机，登陆 GEDU（密码 gedu）后，在桌面点击右键，选择一张不同的壁纸，以便更容易与 GE64 虚拟机区分，按 Ctrl + Alt + T 打可一个终端窗口，执行如下命令启动 gdb：

```
$ cd labs/linux-source-4.8.0/
$ ln -s /usr/lib/debug/boot/vmlinux-4.8.0-36-generic vmlinux
$ stty -F /dev/ttyS0 115200
$ gdb -s vmlinux
```

4. 在 gdb 中执行如下命令设置调试目标：

```
(gdb) target remote /dev/ttyS0
```

5. 切换到 GE64 虚拟机，让其开始启动，顺利的话，它会打印出如下信息，并中断到 gdb

```
KASLR disabled: 'nokaslr' on cmdline.

[    1.059694] KGDB: Waiting for connection from remote gdb...

Entering kdb (current=0xffff88003e27ab80, pid 1) on processor 0 due to Keyboard
Entry
[0]kdb> Supported+;QThreadEvents+;no-resumed+;xmlRegisters=i386#6a$qSupported
```

6. 在 gdb 中，执行如下命令：

```
(gdb) set architecture i386:x86-64  # 设置目标机架构
(gdb) bt                            # 观察栈回溯信息
(gdb) c                             # 让其继续启动
```

7. 登录 GE64，Ctrl+Alt+T 打开一个终端窗口，执行如下命令：

```
$ sudo insmod llaolao.ko                # 加载内核模块
```

8. 执行 dmesg 观察内核的打印信息，可以看到 llaolao 驱动打印的如下信息：

```
[  639.631304] Hi, I am llaolao at address: ffffffffc060f000
```

9. 执行如下命令记录下 llaolao 模块在内存中的地址：

```
$ cat /sys/module/llaolao/sections/.text
0xffffffffc0106000
$ cat /sys/module/llaolao/sections/.bss
0xffffffffc01084c0
$ cat /sys/module/llaolao/sections/.data
0xffffffffc0108000
```

** 最好把上诉地址复制到主机上的记事本中，稍后要在 GD64 虚拟机中使用

10. 继续在终端窗口执行如下命令，启动模拟错误 daemon 的应用程序：

```
$ cd /home/gedu/labs/filetrap           # 切换到应用程序文件夹
$ sudo ./filetrap                       # 一定要以 sudo 方式启动
```

11. 按回车键，回到命令提示符(filetrap 内部会创建子进程继续工作),执行如下命令，触发 GE64 内核中断到 gdb

```
$ sudo su
$ echo g > /proc/sysrq-trigger
```

12. 切换到 GD64 虚拟机，在 gdb 中执行如下命令加载 llaolao 模块的符号文件：

```
(gdb) add-symbol-file llaolao.ko 0xffffffffc0106000 -s .data 0xffffffffc0108000
-s .bss 0xffffffffc01084c0
```

当 gdb 询问是否加载时，确认信息无误后，按 y 键

13. 在 gdb 中执行如下命令，查找虚拟文件的写回调函数，并对其设置断点：

```
(gdb) info function huadeng             # 列出包含 huadeng 字样的函数
(gdb) b huadeng_write                   # 对写回调函数设置断点
(gdb) c                                 # 恢复目标机执行
```

14. 断点应该很快会命中，显示如下信息：

Thread 384 hit Breakpoint 1, huadeng_write (file=0xffff88003a279100,
buffer=0x7ffc6294cc80 "64", length=2, offset=0xffff880035507f18)
at /home/gedu/labs/llaolao/main.c:143

参数 buffer 的内容为 64，代表十进制 100，这是正常的写动作

15. 执行 bt 命令观察执行经过

```
#0  huadeng_write (file=0xffff88003a279100, buffer=0x7ffc6294cc80 "64", length=2,
    offset=0xffff880035507f18) at /home/gedu/labs/llaolao/main.c:143
#1  0xffffffff81232788 in __vfs_write (file=<optimized out>, p=<optimized out>,
    count=<optimized out>, pos=<optimized out>)
    at /build/linux-hwe-eyfT8D/linux-hwe-4.8.0/fs/read_write.c:510
#2  0xffffffff81232ed8 in vfs_write (file=0xffff88003a279100, buf=0x7ffc6294cc80 "64",
    count=2, pos=0xffff880035507f18)
    at /build/linux-hwe-eyfT8D/linux-hwe-4.8.0/fs/read_write.c:584
#3  0xffffffff81234335 in SYSC_write (count=<optimized out>, buf=<optimized out>,
    fd=<optimized out>) at /build/linux-hwe-eyfT8D/linux-hwe-4.8.0/fs/read_write.c:631
#4  SyS_write (fd=<optimized out>, buf=140721962404992, count=2)
    at /build/linux-hwe-eyfT8D/linux-hwe-4.8.0/fs/read_write.c:623
#5  0xffffffff81896576 in entry_SYSCALL_64 ()
    at /build/linux-hwe-eyfT8D/linux-hwe-4.8.0/arch/x86/entry/entry_64.S:207
#6  0x0000000000000000 in ?? ()
```

16. 执行 c 命令恢复目标执行，断点会再次命中，显示如下信息：

Thread 384 hit Breakpoint 1, huadeng_write (file=0xffff88003a279100,
buffer=0x18bf250 "sent 64 to driver\nreturn 2 with errno 9\n\n", length=18,
offset=0xffff880035507f18) at /home/gedu/labs/llaolao/main.c:143

** 注意，这就是那个邪恶的写操作，本来是 log 信息，错误地写给了设备文件，这样一写就把刚才写的 64 覆盖掉了。请结合 daemon 中的如下代码来理解错误原因：

```
//
// business logic run here
//
do
{
    read(fd_device, buffer, 2);
    buffer[2]=0;

    gauge = calc_gauge(atoi(buffer)); //

    sprintf(buffer, "%d", gauge);

    write(fd_device, buffer, 2);

    d4d(0, "sent %s to driver\n", buffer);
    sleep(1);
}while(1);
```

17. 执行 c 命令恢复目标执行，会发现上诉两个写动作交替地循环发生。

```
Thread 384 hit Breakpoint 1, huadeng_write (file=0xffff88003a279100,
    buffer=0x7ffc6294cc80 "64", length=2, offset=0xffff880035507f18)
    at /home/gedu/labs/llaolao/main.c:143
143     {
(gdb) c
Continuing.

Thread 384 hit Breakpoint 1, huadeng_write (file=0xffff88003a279100,
    buffer=0x18bf250 "sent 64 to driver\nreturn 2 with errno 9\n\n", length=18,
    offset=0xffff880035507f18) at /home/gedu/labs/llaolao/main.c:143
143     {
```

18. 执行 command 1 命令为断点附加命令，参考如下截图，输入黑体部分内容

(gdb) command 1
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
>print "****write happens******"**
>c

>end

19. 执行 c 命令恢复目标，会看到上面附加的命令自动执行

```
$1 = "******write happens******"

Thread 384 hit Breakpoint 1, huadeng_write (file=0xffff88003a279100,
    buffer=0x7ffc6294cc80 "64", length=2, offset=0xffff880035507f18)
    at /home/gedu/labs/llaolao/main.c:143
143     {
$2 = "******write happens******"

Thread 384 hit Breakpoint 1, huadeng_write (file=0xffff88003a279100,
    buffer=0x18bf250 "sent 64 to driver\nreturn 2 with errno 9\n\n", length=18,
    offset=0xffff880035507f18) at /home/gedu/labs/llaolao/main.c:143
143     {
```

输出的信息满一屏幕后，gdb 会征询意见，输入回车会让目标继续执行，输入 q 会到 gdb 的命令接口

20. 尝试讲座中提到的其它命令，然后执行 quit 命令中止调试会话，退出 gdb

试验 3：观察/proc/meminfo 输出变化，理解其含意

试验目的：观察 LINUX 内核/proc/meminfo 输出信息，理解其意义

试验环境：GE64 虚拟机

启动和登陆 GE64 虚拟机，使用 Ctrl + Alt + T 热键打开一个终端窗口

a) MemTotal: Total usable ram (i.e. physical ram minus a few reserved bits and the kernel binary code)

```
$ cat /proc/meminfo   #记录当前的 MemTotal 值
```

重新启动，启动时按住 Shift 键，调出 Grub 菜单，选择 GEDU Kernel with Memory Checker 选项，加载带有补丁的内核（boot 时分配内存），然后记录 MemTotal 的新值

Q: alloc_bootmem() 分配的内存是否计入 MemTotal?
   何时能调用 alloc_bootmem() ?

b) MemFree: The sum of LowFree + HighFree

记录在 global_page_state(NR_FREE_PAGES)

```
$ cd labs/driver
$ sudo insmod gedu_mem.ko max_pages=1024000 #allocate max 1024000 pages
$ cd /sys/devices/gedu_mem
$ sudo su #提升权限，输入 gedu 密码
$ cat /proc/meminfo | grep MemFree #记录 MemFree，比如 147596 kB
$ echo A1000 > test       #allocate 100 pages
$ cat /proc/meminfo | grep MemFree #检查 MemFree 变化，比如 144000 kB
$ echo F1000 > test       #Free 100 pages
```

Q: 如果 MemFree 变少，是否能知道减少的内存被谁用了吗?

c) MemAvailable: An estimate of how much memory is available for starting new applications, without swapping. Calculated from MemFree, SReclaimable, the size of the file LRU lists, and the low watermarks in each zone.The estimate takes into account that the system needs some page cache to function well, and that not all reclaimable slab will be reclaimable, due to items being in use. The

impact of those factors will vary from system to system.
在 si_mem_available()中计算

- Estimate the amount of memory available for userspace allocations, without causing swapping.

global_page_state(NR_FREE_PAGES) - totalreserve_pages

- Not all the page cache can be freed, otherwise the system will start swapping. Assume at least half of the page cache, or the low watermark worth of cache, needs to stay.

pages[LRU_ACTIVE_FILE] + pages[LRU_INACTIVE_FILE]
- min(pagecache / 2, wmark_low)

- Part of the reclaimable slab consists of items that are in use, and cannot be freed. Cap this estimate at the low watermark.

global_page_state(NR_SLAB_RECLAIMABLE) -
min(global_page_state(NR_SLAB_RECLAIMABLE) / 2, wmark_low)
按照 b）分配释放 page，检查 MemAvailable 改变

d) Buffers: Relatively temporary storage for raw disk blocks shouldn't get tremendously large (20MB or so)
在 nr_blockdev_pages()中计算

```
list_for_each_entry(bdev, &all_bdevs, bd_list) {
 ret += bdev->bd_inode->i_mapping->nrpages;
}
```

检查执行该命令前后 Buffers 的变化

```
$ sudo dd if=/dev/sda of=dump.bin bs=1M count=30
```

e) Cached: in-memory cache for files read from the disk (the pagecache). Doesn't include SwapCached

global_page_state(NR_FILE_PAGES) - total_swapcache_pages()
- nr_blockdev_pages()
检查执行如下命令前后 Cached 变化

```
gedu@gedu-VirtualBox:~/labs/app$ cat /proc/meminfo | grep Cached
Cached:           292828 kB
SwapCached:         1404 kB

$ ./gedu_memtest1

gedu@gedu-VirtualBox:~/labs/app$ cat /proc/meminfo | grep Cached
Cached:           305316 kB
SwapCached:         3048 kB
```

f) SwapCached: Memory that once was swapped out, is swapped back in but still also is in the swapfile (if memory is needed it doesn't need to be swapped out AGAIN because it is already in the swapfile. This saves I/O)
在 total_swapcache_pages()中计算

for (i = 0; i < MAX_SWAPFILES; i++) ret += swapper_spaces[i].nrpages;

g) Active: Memory that has been used more recently and usually not reclaimed unless absolutely necessary

pages[LRU_ACTIVE_ANON] + pages[LRU_ACTIVE_FILE]

- Active(anon): pages[LRU_ACTIVE_ANON])
- Active(file): pages[LRU_ACTIVE_FILE]

检查执行如下命令前后 Active 变化，以及 Active(anon) + Active(file)和 Active 的关系

$ ./gedu_memtest2

h) Inactive: Memory which has been less recently used. It is more eligible to be reclaimed for other purposes

pages[LRU_INACTIVE_ANON] + pages[LRU_INACTIVE_FILE]

- Inactive(anon): pages[LRU_INACTIVE_ANON]
- Inactive(file): pages[LRU_INACTIVE_FILE]

检查以上 3 者之间的关系

i) Unevictable: pages[LRU_UNEVICTABLE]

j) Mlocked: global_page_state(NR_MLOCK)

k) SwapTotal: total amount of swap space available

si_swapinfo(): total_swap_pages + nr_to_be_unused

l) SwapFree: Memory which has been evicted from RAM, and is temporarily on the disk

atomic_long_read(&nr_swap_pages) + nr_to_be_unused

m) Dirty: Memory which is waiting to get written back to the disk

global_page_state(NR_FILE_DIRTY)

n) Writeback: Memory which is actively being written back to the disk

global_page_state(NR_WRITEBACK)

o) AnonPages: Non-file backed pages mapped into userspace page tables

global_page_state(NR_ANON_PAGES)

p) Mapped: files which have been mmaped, such as libraries

global_page_state(NR_FILE_MAPPED)

q) Shmem: *global_page_state(NR_SHMEM)*

r) Slab: in-kernel data structures cache

global_page_state(NR_SLAB_RECLAIMABLE) +
global_page_state(NR_SLAB_UNRECLAIMABLE)

s) SReclaimable: Part of Slab, that might be reclaimed, such as caches

global_page_state(NR_SLAB_RECLAIMABLE)

t) Sunreclaim: Part of Slab, that cannot be reclaimed on memory pressure

global_page_state(NR_SLAB_UNRECLAIMABLE)

u) KernelStack: *global_page_state(NR_KERNEL_STACK) * THREAD_SIZE / 1024*

v) PageTables: amount of memory dedicated to the lowest level of page tables.

global_page_state(NR_PAGETABLE)

w) NFS_Unstable: NFS pages sent to the server, but not yet committed to stable storage

global_page_state(NR_UNSTABLE_NFS)

x) Bounce: Memory used for block device "bounce buffers"

global_page_state(NR_BOUNCE)

y) WritebackTmp: Memory used by FUSE for temporary writeback buffers

z) CommitLimit: Based on the overcommit ratio ('vm.overcommit_ratio'), this is the total amount of memory currently available to be allocated on the system. This limit is only adhered to if strict overcommit accounting is enabled (mode 2 in 'vm.overcommit_memory'). The CommitLimit is calculated with the following formula:

*CommitLimit = ([total RAM pages] - [total huge TLB pages]) * overcommit_ratio / 100 + [total swap pages]*

For example, on a system with 1G of physical RAM and 7G of swap with a `vm.overcommit_ratio` of 30 it would yield a CommitLimit of 7.3G. For more details, see the memory overcommit documentation in vm/overcommit-accounting.

在 *vm_commit_limit()*中计算: sysctl_overcommit_kbytes

aa) Committed_AS: The amount of memory presently allocated on the system. The committed memory is a sum of all of the memory which has been allocated by processes, even if it has not been "used" by them as of yet. A process which malloc()'s 1G of memory, but only touches 300M of it will show up as using 1G. This 1G is memory which has been "committed" to by the VM and can be used at any time by the allocating application. With strict overcommit enabled on the system (mode 2 in 'vm.overcommit_memory'), allocations which would exceed the CommitLimit (detailed above) will not be permitted. This is useful if one needs to guarantee that processes will not fail due to lack of memory once that memory has been successfully allocated.

__vm_enough_memory(), *vm_committed_as*

bb) VmallocTotal: total size of vmalloc memory area

Documentation/x86/x86_64/mm.txt

VMALLOC_TOTAL = VMALLOC_END - VMALLOC_START

= 0xffffe8fffffffff - 0xffffc90000000000

cc) VmallocUsed: amount of vmalloc area which is used, fixed to 0

dd) VmallocChunk: largest contiguous block of vmalloc area which is free, fixed to 0

ee) HardwareCorrupted: CONFIG_MEMORY_FAILURE, num_poisoned_pages

ff) AnonHugePages: CONFIG_TRANSPARENT_HUGEPAGE, Non-file backed huge pages mapped into userspace page tables

gg) CmaTotal:

hh) CmaFree:

ii) HugePages_Total: the size of the pool of huge pages.

jj) HugePages_Free: the number of huge pages in the pool that are not yet allocated

kk) HugePages_Rsvd: the number of huge pages for which a commitment to allocate from the pool has been made, but no allocation has yet been made. Reserved huge pages guarantee that an application will be able to allocate a huge

page from the pool of huge pages at fault time.

ll) HugePages_Surp: the number of huge pages in the pool above the value in /proc/sys/vm/nr_hugepages. The maximum number of surplus huge pages is controled by /proc/sys/vm/nr_overcommit_hugepages.

mm) Hugepagesize: 2048 kB

nn) DirectMap4k:

oo) DirectMap2M:

pp) DirectMap1G:

x86-specific, indication of the TLB load

检查执行如下命令观察 OOMkiller

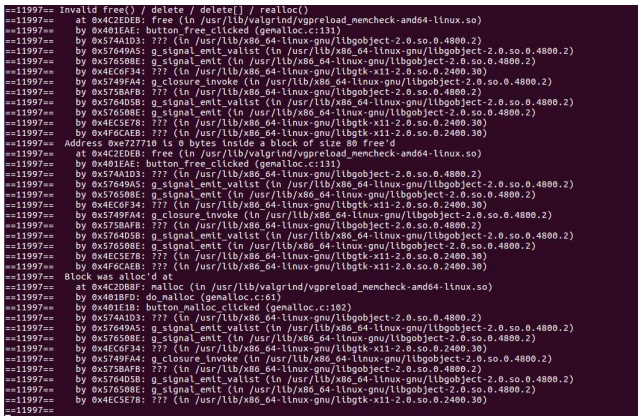
$ ./gedu_memtest3

试验 4：使用 valgrind 调试典型的堆错误

实验目的：使用 valgrind 调堆有关的问题，包括溢出、多次释放和野指针

实验环境：GE64 虚拟机

1. 在终端窗口执行如下命令，启动 GeMalloc 程序
   $ cd /home/gedu/labs/gemalloc
   $ ./gemalloc
2. 点击 Malloc 按钮从堆上分配内存
3. 点击 free 按钮一次，释放 g_lastmalloc 指针所指向的堆块，再点击 free 一次，模拟多重分配错误
4. 因为堆错误有很大随机性，刚才的多次释放可能导致 gemalloc 突然消失，终端窗口出现如下消息，如果没有出错，则重复上面两个步骤几次
   (gemalloc:11931): GLib-ERROR **: /build/glib2.0-prJhLS/glib2.0-2.48.2/./glib/gmem.c:165: failed to allocate 45953224224 bytes
   Trace/breakpoint trap (core dumped)
5. 执行命令$ valgrind --tool=memcheck ./gemalloc 在 valgrind 的监视下运行 gemalloc，重复上面的过程做多重释放，注意观察终端窗口，会看到第二次释放时，valgrind 检测到错误



上面的信息分成三个部分：第二次错误释放的信息，上次释放的过程，最初分配的过程

6. 如果程序退出，则重复第 5 步的命令重新运行，点击分配后，在 Number 标签下的编辑框中输入 100，然后点击 Overflow 触发溢出访问，会看到 valgrind 检测到非法的写动

作

```
==12066== Invalid write of size 8
==12066==    at 0x4C3453F: memset (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==12066==    by 0x401F4B: button_overflow_clicked (gemalloc.c:146)
==12066==    by 0x574A1D3: ??? (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.4800.2)
==12066==    by 0x57649A5: g_signal_emit_valist (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.4800.2)
==12066==    by 0x576508E: g_signal_emit (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.4800.2)
==12066==    by 0x4EC6F34: ??? (in /usr/lib/x86_64-linux-gnu/libgtk-x11-2.0.so.0.2400.30)
==12066==    by 0x5749FA4: g_closure_invoke (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.4800.2)
==12066==    by 0x575BAFB: ??? (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.4800.2)
==12066==    by 0x5764D5B: g_signal_emit_valist (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.4800.2)
==12066==    by 0x576508E: g_signal_emit (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.4800.2)
==12066==    by 0x4EC5E78: ??? (in /usr/lib/x86_64-linux-gnu/libgtk-x11-2.0.so.0.2400.30)
==12066==    by 0x4F6CAEB: ??? (in /usr/lib/x86_64-linux-gnu/libgtk-x11-2.0.so.0.2400.30)
==12066==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

7. 如果程序退出，则重复第 5 步的命令重新运行，在 Number 标签下的编辑框中输入 100，点击 Wildpointer 按钮，触发下面的代码：

```
g_databuffer[element] = 0x88888888;
```

```
==12076== Invalid write of size 4
==12076==    at 0x401FB0: button_wild_clicked (gemalloc.c:157)
==12076==    by 0x574A1D3: ??? (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.4800.2)
==12076==    by 0x57649A5: g_signal_emit_valist (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.4800.2)
==12076==    by 0x576508E: g_signal_emit (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.4800.2)
==12076==    by 0x4EC6F34: ??? (in /usr/lib/x86_64-linux-gnu/libgtk-x11-2.0.so.0.2400.30)
==12076==    by 0x5749FA4: g_closure_invoke (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.4800.2)
==12076==    by 0x575BAFB: ??? (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.4800.2)
==12076==    by 0x5764D5B: g_signal_emit_valist (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.4800.2)
==12076==    by 0x576508E: g_signal_emit (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.4800.2)
==12076==    by 0x4EC5E78: ??? (in /usr/lib/x86_64-linux-gnu/libgtk-x11-2.0.so.0.2400.30)
==12076==    by 0x4F6CAEB: ??? (in /usr/lib/x86_64-linux-gnu/libgtk-x11-2.0.so.0.2400.30)
==12076==    by 0x5749FA4: g_closure_invoke (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.4800.2)
==12076==  Address 0xe2ed950 is 32 bytes before a block of size 48 in arena "client"
==12076==
```

8. 执行 gedit gemalloc.c &打开源代码，理解上诉错误的原因

试验 5：理解系统 Oops 和 Panic

实验目的：学习如何解读 Oops 信息，理解 Panic 过程，分析导致 Panic 的原因

实验环境：GE64 虚拟机

1. 在 GE64 虚拟机的终端窗口执行如下命令，检查当前启动项是否启用了内核调试

   $ cat /proc/cmdline

   ** 如果命令行中包含 kgdboc 等内核调试选项，则需要重启虚拟机，以默认选项启动
2. 在终端窗口输入如下命令，加载 llaolao.ko

   $ cd labs/llaolao

   $ sudo insmod llaolao.ko　　# 加载驱动

   $ cat /proc/llaolao　　# 观察虚文件
3. 继续执行如下命令，触发 llaolao 驱动内的除零操作

   $ sudo su　　# 提升权限，密码 gedu

   $ echo “div0”>/proc/llaolao　　# 写虚文件，触发除零异常
4. 执行 dmesg 观察系统消息，可以看到下图所示的 Oops 输出，表明刚才的除零动作引发系统打喷嚏（oops），系统没有重启，说明内核认为不够严重，没有进入 Panic 过程

根据讲义，认真理解上面的 Oops 信息，遇到疑难，请向讲师提问

5. 继续执行如下命令，触发 llaolao 驱动内的访问空指针操作

   $ echo “nullp”>/proc/llaolao　　# 写虚文件，触发空指针异常

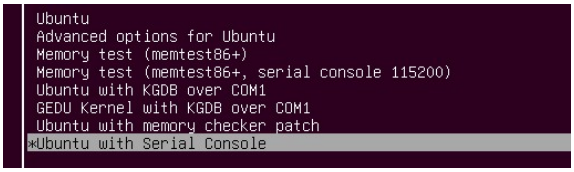
6. 系统应该没有死掉，说明发生在内核模块里普通情况下的空指针也没有升级为 Panic，执行 dmesg 观察系统信息，看到的信息类似下图，仔细理解其中的信息

7. 继续执行如下命令，启动 llaolao 驱动内的计时器，在计时器内访问空指针

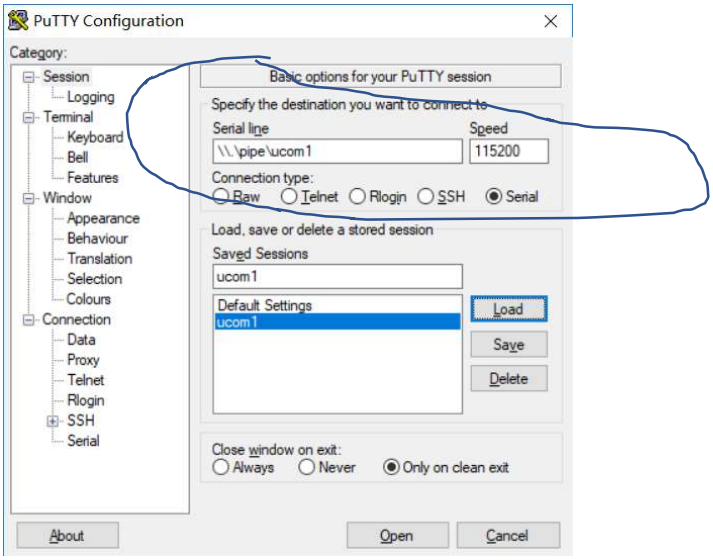
   $ echo “timer0”>/proc/llaolao　　# 写虚文件，启动计时器

   计时器定时 5 秒后会在中断上下文下访问空指针，系统应该不能动弹了…

8. 点击 VirtualBox 窗口的菜单“控制 > 重启”，强制重启 GE64 虚拟机，重启时记住按住 Shift 按键，调出 Grub 菜单，选择 Ubuntu with Serial Console 选项，然后按回车启动



9. 本步骤需要一个小的串口终端程序，对于 Windows 系统，请运行 labs\putty\putty.exe，对于 MacOS, 可以使用预装的 Terminal 程序(Applications folder => click on the Utilities folder => then click on Terminal)，以下以 putty 为例
10. 在 putty 中，按如下截图输入串口参数（圈选部分），然后点击 Open



11. 在 GE64 虚拟机中，重复上面的 2-7 步（主要是第 7 步），当 panic 时，从串口终端可以观察到下图所示的 panic 信息（在 end trace 和 end panic 之间的部分）

```
[  221.443768] Code: 00 66 2e 0f 1f 84 00 00 00 00 00 0f 1f 44 00 00 55 48 c7 c6
 80 30 4b c0 48 89 e5 53 48 89 fb 48 c7 c7 a0 32 4b c0 e8 ae c8 6e fb <c7> 03 d0
 ba d0 ba 5b 5d c3 0f 1f 80 00 00 00 00 0f 1f 44 00 00
[  221.528822] RIP  [<ffffffffc04b2050>] ll_timer_callback+0x20/0x30 [llaolao]
[  221.531636]  RSP <ffff8a91ffc03e50>
[  221.539468] CR2: 0000000000000bad
[  221.541840] ---[ end trace 6482fe703e5df570 ]---
[  221.547676] Kernel panic - not syncing: Fatal exception in interrupt
[  221.576064] Kernel Offset: 0x3aa00000 from 0xffffffff81000000 (relocation ran
ge: 0xffffffff80000000-0xffffffffbfffffff)
[  223.967344] ---[ end Kernel panic - not syncing: Fatal exception in interrupt
```

12. 关闭虚拟机和 putty，结束本试验

试验 6：动态打印输出

实验目的：学习使用 LINUX 内核的动态打印信息（dynamic printk）机制，理解其工作原理

实验环境：GE64 虚拟机

1. 启动和登陆 GEDU 虚拟机，使用 Ctrl + Alt + T 热键打开一个终端窗口
2. 执行如下命令切换为超级用户身份，并进入到控制动态信息输出的 debugfs 虚拟目录：

   $ sudo su

   <输入密码 gedu>

   $ cd /sys/kernel/debug/dynamic_debug
3. 执行如下命令观察内核模块有关的动态信息：

   $ cat control | grep "\[module\]"

```
root@gedu-VirtualBox:/sys/kernel/debug/dynamic_debug# cat control | grep "\[module\]"
# filename:lineno [module]function flags format
kernel/module.c:3785 [module]SYSC_finit_module =_ "finit_module: fd=%d, uargs=%p, flags=%i\012"
kernel/module.c:3765 [module]SYSC_init_module =_ "init_module: umod=%p, len=%lu, uargs=%p\012"
kernel/module.c:3125 [module]move_module =_ "\0110x%lx %s\012"
```

   以上输出中，第三列为标志列，其中的=_表示该行信息处于禁止状态
4. 执行如下命令信息启用 module 模块的所有信息：

   $ echo -n 'module module +p' > control
5. 重复第三步，确认标志部分已经改变为=p:

```
root@gedu-VirtualBox:/sys/kernel/debug/dynamic_debug# cat control | grep "\[module\]"
# filename:lineno [module]function flags format
kernel/module.c:3785 [module]SYSC_finit_module =p "finit_module: fd=%d, uargs=%p, flags=%i\012"
kernel/module.c:3765 [module]SYSC_init_module =p "init_module: umod=%p, len=%lu, uargs=%p\012"
kernel/module.c:3125 [module]move_module =p "\0110x%lx %s\012"
kernel/module.c:3106 [module]move_module =p "final section addresses:\012"
```

6. 执行如下命令找到 hio 驱动，然后再加载这个驱动

   $ locate hio.ko

   $ insmod /usr/lib/debug/lib/modules/4.8.0-36-generic/kernel/ubuntu/hio/hio.ko

   [可以用鼠标选中文件全路径，然后点击滚轮进行粘贴]

7. 执行 dmesg 观察刚才动作所触发的信息输出：

   $ dmesg

```
root@gedu-VirtualBox:/sys/kernel/debug/dynamic_debug# dmesg
[33664.973950] finit_module: fd=3, uargs=0000561ffbd68246, flags=0
[33665.019761] Core section allocation order:
[33665.019763]  .text
[33665.019764]  .text.unlikely
[33665.019765]  .exit.text
[33665.019765]  .note.gnu.build-id
```

8. 执行如下命令序列，先卸载驱动，然后关闭信息输出，再加载驱动，验证关闭信息后，加载驱动也不会产生打印信息

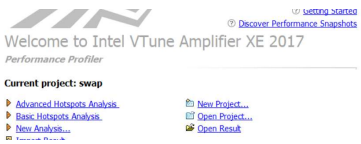
```
$ dmesg -c                                          #清理 printk 缓冲区
$ rmmod hio                                         #卸载驱动
$ echo -n "func SYSC_finit_module -p" > control     #关闭指定函数的信息打印
$ insmod hio                                        #加载驱动
$ dmesg                                             #观察 printk 缓冲区
```

试验 7：使用 vTune 分析应用程序的执行热点

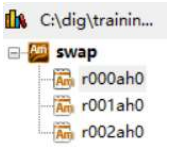
实验目的：使用 vTune 图形分析工具进行分析， 学习不同分析视图的用法，理解 vTune 中的关键性能指标

实验环境：Windows 主机

1. 在 C 盘建立 labs 子目录，将试验优盘上的 vtune.zip 复制到 c:\labs，然后解压缩到这个目录中。
2. 如果还没有安装 VTUNE，那么请安装，安装后，在开始菜单找到链接（Intel VTune Amplifier…），启动 VTUNE 的主程序。
3. 在 Welcome 页面上点击 Open Project…，然后打开 c:\labs\swap\swap.amplxeproj



4. 在项目导航（Project Navigator）子窗口中，保存了三次调优记录，名字中的 r00n 代表第 n 次 run，ah 代表分析类型为 advanced hotspot（高级热点），选择其中的第一个，vtune 会加载当时采样的数据，需要几秒到几十秒时间



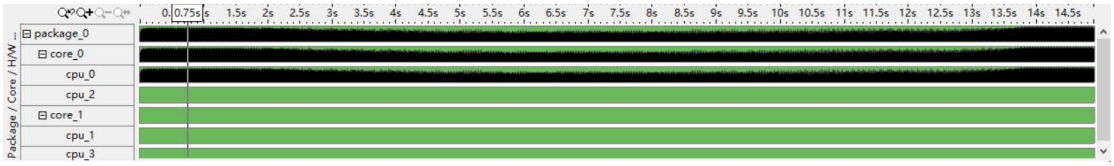
5. 加载完成后，VTune 会在窗口主区域产生多个视图页面，选择 Event Count 页面，按 Core/Thread/Function/Call Stack 组合，可以看到硬件事件的分布情况

Grouping: Core / Thread / Function / Call Stack

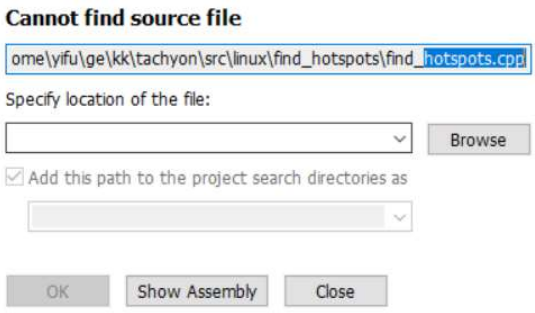
Core / Thread / Function / Call Stack	Hardware Event Count by Hardware Event Type				
	INST_RETIRED.ANY	CPU_CLK_UNHALTED.THREAD	CPU_CLK_UNHALTED.REF_TSC	Syscalls	Sysc
core_0	80,442,805,706	37,077,494,200	37,069,087,250	0	
tachyon_find_ho (TID: 5258)	80,442,805,706	37,077,494,200	37,069,087,250	0	
initialize_2D_buffer	49,189,369,768	20,628,270,394	20,623,602,125	0	
grid_intersect	16,592,883,620	8,803,943,629	8,801,943,125	0	
sphere_intersect	11,185,275,861	5,938,512,034	5,937,160,200	0	
grid_bounds_intersect	1,412,046,177	740,754,514	740,586,475	0	

上图表明 initialize_2D_buffer 函数内发生的 INST_RETIRED（指令老化）事件最多，说明 CPU 在这个函数上执行的指令书最多

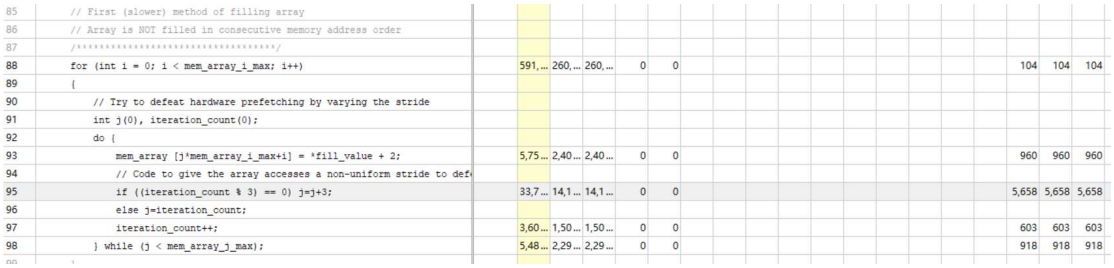
6. 观察页面底部，可以看到不同事件展示视图，比如下面的截图说明选择的硬件事件都发生在 0 号 CPU 上



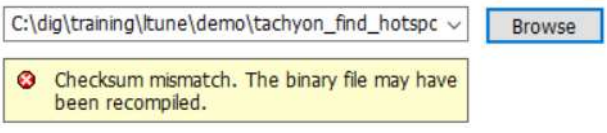
7. 双击 initialize_2D_buffer 函数，VTUNE 会尝试自动打开源程序文件，但是自动寻找失败。点击 Browse，指点 c:\labs\swap\find_hotspots.cpp



8. 成功指定源文件后，VTUNE 会展示源代码视图，右侧的数字为事件数量



9. 点击汇编按钮，VTUNE 提示要提供二进制模块，可惜当时没有保存，尝试指定 c:\labs\swap\ tachyon_find_hotspots，VTUNE 会提示校验和不匹配

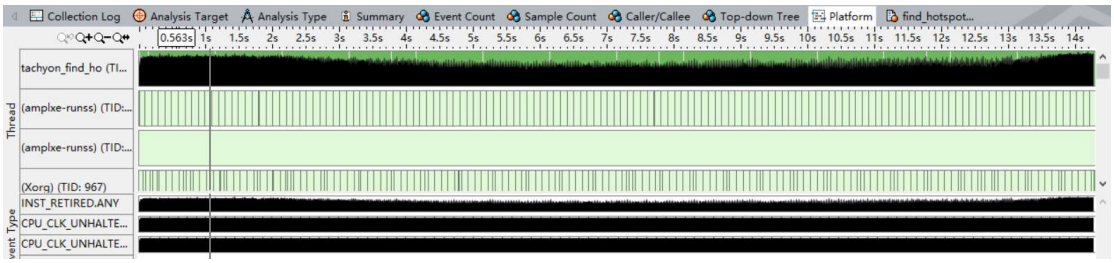


10. 选择 Top-Down Tree 试图，观察热点的执行路径（从父函数到子函数）

Grouping: Call Stack

Function Stack	Hardware Event Count: Total by Hardware Event Type		
	INST_RETIRED.ANY ▼	CPU_CLK_UNHALTED.THREAD	CPU_CLK_UNHALTED.REF_TSC
Total	80,442,805,706	37,077,494,200	37,069,087,250
[Unknown stack frame(s)]	80,442,805,706	37,077,494,200	37,069,087,250
render_one_pixel	49,263,633,630	20,660,804,602	20,656,128,950
initialize_2D_buffer	49,189,369,768	20,628,270,394	20,623,602,125
video::get_color	14,030,675	5,005,193	5,004,075
grid_intersect	18,004,929,797	9,544,698,143	9,542,529,600
sphere_intersect	11,185,275,861	5,938,512,034	5,937,160,200
pos2grid	474,479,642	225,224,010	225,173,250

11. 选择 Platform 视图，可以看到整个系统（包括其它比较活跃的进程）情况



第三行是 Xorg 的 967 号线程，可以明显看到它每隔 100ms 执行一下的周期性特征

12. 尝试其它功能，或者关闭 VTUNE 结束试验

附录 A:

Q: alloc_bootmem() 分配的内存是否计入 MemTotal?

>>从实验结果看，不计入。

何时能调用 alloc_bootmem() ?

>>把 test_allocate_large_mem();移到 kmem_cache_init();后面看看能分多少。

检查如下 API 可知要在 SLAB ready 之前才行。

```
static void * __init ___alloc_bootmem_nopanic() {
        void *ptr;
        if (WARN_ON_ONCE(slab_is_available()))
            return kzalloc(size, GFP_NOWAIT);
        ...
    }
```

Q: 如果 MemFree 变少，是否能知道减少的内存被谁用了吗?

>> /proc/meminfo 以及 sysrq 的 show mem 都没法给出完整的已分配内存使用在什么地方，特别是 driver 调用 alloc_pages 分配的内存，只能看到 MemFree 减少，但是谁用的并不知道。比如网卡的 driver 在接收时会分配 page，但要到上层协议栈设置了 owner 才能用 ss 命令看到。

void skb_set_owner_w(struct sk_buff *skb, struct sock *sk)

为提高系统性能，减少 cache 替换，kernel 使用了 per_cpu_pages 来缓存一些单页的 page，所以分配单页的时候首先从这个 list 里面取，如果 list 为空，会从 buddy 中一次分配一批单页的 page 填充该 list， 然后取一个，释放单页时也同样先放到该 list， 数量到一个比较高的值时才一起释放到 buddy 中 。

```
/* Allocate a page from the given zone. Use pcplists for order-0 allocations. */
    struct page *buffered_rmqueue()
    void free_hot_cold_page()
```

另外 vmstat counters are not perfectly accurate and the estimated value for counters such as NR_FREE_PAGES can deviate from the true value by

[在此处键入]

nr_online_cpus * threshold.

```
        void __mod_zone_freepage_state() {
        __mod_zone_page_state(zone, NR_FREE_PAGES, nr_pages);
        ...
    }
    void __mod_zone_page_state() {
    {
        struct per_cpu_pageset __percpu *pcp = zone->pageset;
        s8 __percpu *p = pcp->vm_stat_diff + item;
        long x;
        long t;
        x = delta + __this_cpu_read(*p);
        t = __this_cpu_read(pcp->stat_threshold);
        if (unlikely(x > t || x < -t)) {
            zone_page_state_add(x, zone, item);
            x = 0;
        }
        __this_cpu_write(*p, x);
    }
```