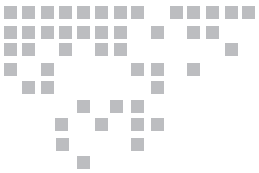

Chapter 2 第 2 章

系统性能分析

2.1 性能分析简介

很多人喜欢把系统性能分析称为性能优化，这里特意避免使用“优化”一词，是因为优化是一个复杂的、基于业务场景的工作，有时候看似不正常的系统性能现象也可能是正常的表现；有时候看似做了很多针对性的参数调整，但是实际效果可能不如硬件性能提升来得明显。所以本章并不会重点讲解如何优化，而是重点讲解如何分析系统性能。

一般在条件有限制的情况下，性能分析主要集中在两个方面：

- 响应时间
- 单位时间效率

本章将通过分析系统 CPU、磁盘、内存来讲解寻找系统与应用热点与瓶颈。

2.2 系统分析的基本工具

2.2.1 CPU 性能分析工具

1. mpstat

mpstat 是报告 CPU 状态的工具，用法比较简单，基本用法如下。

（1）每 1 秒统计一次 CPU 状态，一共统计 3 次

示例代码如下：

```
[root@server ~]# LANG=c
[root@server ~]# mpstat 1 3
```

```
Linux 3.10.0-123.el7.x86_64 (server.example.com)        05/28/15        _x86_64_
    (1 CPU)

02:00:06      CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest
    %gnice   %idle
02:00:07      all    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
    0.00  100.00
02:00:08      all    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
    0.00  100.00
02:00:09      all    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
    0.00  100.00
Average:      all    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
    0.00  100.00
```

在上面的代码中，LANG=c 的目的是将时间从 12 小时制转换成 24 小时制。

（2）查看多核 CPU 的使用情况

示例代码如下：

```
[root@server ~]# mpstat -P ALL 1 1
Linux 3.10.0-123.el7.x86_64 (server.example.com)        05/28/2015      _x86_64_
(4 CPU)

02:07:26 AM  CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest
    %gnice   %idle
02:07:27 AM  all    0.25    0.00    0.00    0.00    0.00    0.00    0.00    0.00
    0.00   99.75
02:07:27 AM    0    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
    0.00  100.00
02:07:27 AM    1    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
    0.00  100.00
02:07:27 AM    2    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
    0.00  100.00
02:07:27 AM    3    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
    0.00  100.00

Average:      CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest
    %gnice   %idle
Average:      all    0.25    0.00    0.00    0.00    0.00    0.00    0.00    0.00
    0.00   99.75
Average:        0    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
    0.00  100.00
Average:        1    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
    0.00  100.00
Average:        2    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
    0.00  100.00
Average:        3    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
    0.00  100.00
```

其中的每一列代表的含义如下：

- %user：用户态程序

- %nice：优先级调整
- %sys：内核态消耗
- %iowait：磁盘等待
- %irp：硬件中断
- %soft：软件中断
- %steal：处理 hyperviosr 的消耗
- %guest：虚拟机消耗掉的 CPU
- %idle：CPU 空闲

更多的解释请查看 man 手册。

2. 查看 CPU 硬件信息的工具

(1) lscpu

lscpu 这个命令是在 CentOS 6 中引入的，在 CentOS 5 上没有此工具。lscpu 可以查看 CPU 的型号、一级缓存、二级缓存等信息。示例代码如下：

```
[root@server ~]# lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 58
Model name:            Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
Stepping:              9
CPU MHz:               3901.000
BogoMIPS:              7802.00
Virtualization:        VT-x
Hypervisor vendor:     VMware
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-3
```

(2) dmidecode

在 CentOS 5 上查看 CPU 硬件信息可以使用 dmidecode 工具，dmidecode 会提供比 lscpu 更为详细的细节信息。示例代码如下：

```
[root@server ~]# dmidecode -t processor | less
# dmidecode 2.12
SMBIOS 2.4 present.

Handle 0x0004, DMI type 4, 35 bytes
Processor Information
    Socket Designation: CPU socket #0
    Type: Central Processor
    Family: Unknown
    Manufacturer: GenuineIntel
    ID: A9 06 03 00 FF FB AB 1F
    Version:          Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
    Voltage: 3.3 V
    External Clock: Unknown
    Max Speed: 30000 MHz
    Current Speed: 3900 MHz
    Status: Populated, Enabled
    Upgrade: ZIF Socket
    L1 Cache Handle: 0x0094
    L2 Cache Handle: 0x0095
    L3 Cache Handle: Not Provided
    Serial Number: Not Specified
    Asset Tag: Not Specified
    Part Number: Not Specified

Handle 0x0005, DMI type 4, 35 bytes
Processor Information
    Socket Designation: CPU socket #1
    Type: Central Processor
    Family: Unknown
    Manufacturer: GenuineIntel
    ID: A9 06 00 00 FF FB AB 1F
    Version:          Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
    Voltage: 3.3 V
    External Clock: Unknown
    Max Speed: 30000 MHz
    Current Speed: 3900 MHz
    Status: Populated, Enabled
    Upgrade: ZIF Socket
    L1 Cache Handle: 0x0096
    L2 Cache Handle: 0x0097
    L3 Cache Handle: Not Provided
    Serial Number: Not Specified
    Asset Tag: Not Specified
    Part Number: Not Specified

Handle 0x0006, DMI type 4, 35 bytes
......
```

2.2.2 内存性能分析工具

1. free

free 是所有系统工程师都会使用的命令，这里要搞清楚 cache、buffer、used 和 total 之间的关系。

首先 total=used+free，这很容易理解。

- total：物理内存的总大小。
- used：被使用的内存大小。
- free：未被使用的内存大小。

以下是 free 输出的结果：

```
[root@server ~]# free -m
    total       used       free     shared    buffers     cached
Mem:           980        337        643          0         15         91
-/+ buffers/cache:        230        750
Swap:         1023          0       1023
```

在 Mem：这行中的 buffers 和 caches 指的是系统已经分配但是还未被使用的 buffers 和 caches。这里为 15+91=106，所以共有 106MB 的 cache/buffer 还未被使用。

-/+ buffers/cache 行中，used 这列代表实际使用的 buffer/cache 总量，即 337–230=107，约等于前面的 106。free 这列代表的是系统真正可以使用的内存。

关于 cache 与 buffer 的区别，在后面的章节会讲到。

2. /proc/meminfo

meminfo 里包含了所有的内存相关信息。示例代码如下：

```
[root@server ~]# cat /proc/meminfo
MemTotal:        1519556 kB
MemFree:         1132324 kB
MemAvailable:    1192320 kB
Buffers:            1120 kB
Cached:           169944 kB
SwapCached:            0 kB
Active:           124640 kB
Inactive:         139784 kB
Active(anon):      93992 kB
Inactive(anon):     8376 kB
Active(file):      30648 kB
Inactive(file):   131408 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:       2113532 kB
SwapFree:        2113532 kB
Dirty:                 0 kB
Writeback:             0 kB
```

```
AnonPages:              93468 kB
Mapped:                 30460 kB
Shmem:                   9008 kB
Slab:                   50548 kB
SReclaimable:           21124 kB
SUnreclaim:             29424 kB
KernelStack:             4920 kB
PageTables:              8332 kB
NFS_Unstable:               0 kB
Bounce:                     0 kB
WritebackTmp:               0 kB
CommitLimit:          2873308 kB
Committed_AS:          411276 kB
VmallocTotal:    34359738367 kB
VmallocUsed:           187720 kB
VmallocChunk:    34359533052 kB
HardwareCorrupted:          0 kB
AnonHugePages:         22528 kB
HugePages_Total:            0
HugePages_Free:             0
HugePages_Rsvd:             0
HugePages_Surp:             0
Hugepagesize:            2048 kB
DirectMap4k:            65408 kB
DirectMap2M:          1507328 kB
```

其中一些参数的含义会在后面的章节提到。

3. vmstat

可以说 vmstat 是所有系统管理员必会的命令之一，vmstat 的用法与 mpstat 类似。但是 vmstat 提供了非常丰富的系统信息。因此需要对输出内容有很清楚的了解。下面将讲解几个重点输出，示例如下：

```
[root@server ~]# vmstat -a 1 5
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
    r  b   swpd   free  inact active   si   so    bi    bo   in   cs us sy id  wa st
    1  0      0 1055176 155716 130904    0    0     4     1    9   15  0  0 100  0  0
    0  0      0 1055144 155716 130980    0    0     0     0   44   80  0  0 100  0  0
    0  0      0 1055144 155716 130980    0    0     0     0   30   50  0  0 100  0  0
    0  0      0 1055144 155716 130980    0    0     0     0   34   54  0  0 100  0  0
    0  0      0 1055144 155716 131000    0    0     0     0   24   39  0  0 100  0  0
```

对于其中部分输出项的说明如下。

（1）procs

在 procs 中，b 这列表示的是不可中断睡眠的进程，这个数值往往与磁盘 I/O 有关。

（2）system

system 这列中有两列，分别是 in 和 cs。

in 代表的是每秒钟的中断次数，包括时钟中断。何为时钟中断？时钟中断指的是系统向 CPU 发出信号，请求处理新的时间片。请求的频率叫做时钟频率。这个参数是在内核中配置的。默认配置是每秒钟 1000 次，相当于 1 毫秒一次。这个参数值出现在 /boot/config-{kernel-version} 中，可以使用 grep 命令查看到系统当前的数值，示例代码如下：

```
[root@server ~]# grep HZ /boot/config-2.6.32-431.el6.x86_64
CONFIG_NO_HZ=y
# CONFIG_HZ_100 is not set
# CONFIG_HZ_250 is not set
# CONFIG_HZ_300 is not set
CONFIG_HZ_1000=y
CONFIG_HZ=1000
```

cs 代表的是每秒上下文切换数。何为上下文切换？当 CPU 收到时钟请求去处理下一个时间片里的进程时，即处理下一个进程缓存在 CPU 一级缓存的数据，这就是上下文切换。

in 与 cs 数值偏高说明系统非常繁忙。

（3）CPU

CPU 这部分中，st 这列往往会被很多人忽视，其实这列在虚拟化的环境中是比较重要的。st 全称是 steal time，指的是强制等待虚拟 CPU 的时间，如果这个数值过高，说明 hypervisor 进程正在为别的虚拟机服务，此时需要等待 hypervisor。在生产环境中 st 持续偏高，说明物理主机上运行了太多的虚拟机，已经超出了物理机器的资源。

2.2.3 磁盘性能分析工具

1. iostat

iostat 也是所有系统管理员必会的命令之一，具体使用不多细说，但可能很多人并不清楚 iostat 输出值的单位是什么含义，而这恰恰是非常重要的。

默认情况下 iostat 输出是以 block 为单位的。以 Blk 开头的值都是以 block 为单位的，在 iostat 中，一个 block 是 512 个字节。示例代码如下：

```
[root@server ~]# iostat 1 5 /dev/sda
Linux 2.6.32-431.11.2.el6.x86_64 (server.example.com) 05/28/2015 _x86_64_ (16 CPU)

avg-cpu:   %user   %nice   %system   %iowait   %steal   %idle
           5.62    0.00      4.84      0.80     0.00   88.74

Device:    tps    Blk_read/s    Blk_wrtn/s       Blk_read     Blk_wrtn
sda      24.02        442.14        683.54   14196546071  21947219860

avg-cpu:   %user   %nice   %system   %iowait   %steal   %idle
           6.92    0.00      4.85      0.06     0.00   88.17

Device:    tps    Blk_read/s    Blk_wrtn/s    Blk_read    Blk_wrtn
sda       2.00          0.00         24.00           0          24
```

```
avg-cpu:   %user   %nice   %system   %iowait   %steal   %idle
           6.75    0.00      4.73      0.00     0.00   88.52

Device:   tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
sda      0.00         0.00         0.00          0          0

avg-cpu:   %user   %nice    %system   %iowait   %steal   %idle
           8.12    0.00       4.91      0.38     0.00   86.60

Device:   tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
sda      5.00         0.00       112.00          0        112

avg-cpu:   %user   %nice   %system   %iowait   %steal   %idle
           2.46    0.00      4.93      1.77     0.00   90.84

Device:   tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
sda     42.00         0.00       528.00          0        528
```

所以如果希望以 KB 形式显示，需要加上 -k 参数将其转换成字节，如下：

```
[root@server ~]# iostat 1 5 -k /dev/sda
Linux 2.6.32-431.11.2.el6.x86_64 (server.example.com) 05/28/2015 _x86_64_ (16 CPU)

avg-cpu:   %user   %nice   %system   %iowait   %steal   %idle
           5.62    0.00      4.84      0.80     0.00   88.74

Device:   tps   kB_read/s   kB_wrtn/s      kB_read       kB_wrtn
sda     24.02      221.07      341.77   7098281615   10973621830

avg-cpu:   %user   %nice   %system   %iowait   %steal   %idle
           0.75    0.00      5.75      0.00     0.00   93.50

Device:   tps   kB_read/s   kB_wrtn/s   kB_read   kB_wrtn
sda      0.00        0.00        0.00         0         0

avg-cpu:   %user   %nice   %system   %iowait   %steal   %idle
           0.57    0.00      5.28      0.00     0.00   94.15

Device:   tps   kB_read/s   kB_wrtn/s   kB_read   kB_wrtn
sda      4.00        0.00       16.00         0        16

avg-cpu:   %user   %nice   %system   %iowait   %steal   %idle
           0.76    0.00      4.79      0.50     0.00   93.95

Device:   tps   kB_read/s   kB_wrtn/s   kB_read   kB_wrtn
sda      9.00        0.00      100.00         0       100

avg-cpu:   %user   %nice   %system   %iowait   %steal   %idle
           1.07    0.00      4.91      1.20     0.00   92.82
```

```
Device:    tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda      34.00         0.00       456.00          0        456
```

2. iotop

iotop 是用 Python 写的一个类似于 top 命令的软件，用来监控磁盘 I/O 的情况。iotop 可以实时监控到每个进程及线程的磁盘读写和 I/O 请求。

其中，需要注意以下几个参数：

- -o：只显示有 I/O 操作的进程和线程。
- -P：只显示进程数。默认是显示进程和线程。
- -k：以千字节显示，更为友好的输出。

示例代码如下：

```
[root@server ~]# iotop -h
Usage: /usr/sbin/iotop [OPTIONS]

DISK READ and DISK WRITE are the block I/O bandwidth used during the sampling
period. SWAPIN and IO are the percentages of time the thread spent respectively
    while swapping in and waiting on I/O more generally. PRIO is the I/O priority
    at which the thread is running (set using the ionice command).

Controls: left and right arrows to change the sorting column, r to invert the
    sorting order, o to toggle the --only option, p to toggle the --processes
option, a to toggle the --accumulated option, i to change I/O priority, q to
    quit, any other key to force a refresh.

Options:
    --version             show program's version number and exit
    -h, --help            show this help message and exit
    -o, --only            only show processes or threads actually doing I/O
    -b, --batch           non-interactive mode
    -n NUM, --iter=NUM    number of iterations before ending [infinite]
    -d SEC, --delay=SEC   delay between iterations [1 second]
    -p PID, --pid=PID     processes/threads to monitor [all]
    -u USER, --user=USER  users to monitor [all]
    -P, --processes       only show processes, not all threads
    -a, --accumulated     show accumulated I/O instead of bandwidth
    -k, --kilobytes       use kilobytes instead of a human friendly unit
    -t, --time            add a timestamp on each line (implies --batch)
    -q, --quiet           suppress some lines of header (implies --batch)
```

2.2.4 sar

sar 可以说是系统性能诊断的瑞士军刀了，它可以提供几乎所有的系统信息。同时也可通过结合 cronjob、sar 记录下系统的实时状态，以便系统管理员能够对过去的性能状态进行分析排错。

1. 自动收集系统活动信息

在 /etc/cron.d/sysstat 里有两个计划任务，sa1 收集当前系统的信息，sa2 汇集当天的系统信息。在 cronjob 里设定了 sa1 每 10 分钟运行一次，sa2 则在每天的 23 点 59 分运行一次。这里建议采用默认值。示例如下：

```
[root@server ~]# cat /etc/cron.d/sysstat
# Run system activity accounting tool every 10 minutes
*/10 * * * * root /usr/lib64/sa/sa1 1 1
# 0 * * * * root /usr/lib64/sa/sa1 600 6 &
# Generate a daily summary of process accounting at 23:53
53 23 * * * root /usr/lib64/sa/sa2 -A
```

这里，sa1 调用了 sadc 来收集当前系统的活动信息，以 2 进制形式保存数据。sadc 叫做数据收集实用程序，它是 sar 的后端程序。

sa2 调用了 sar 来生成当天的系统信息报告，以文本形式保存。

sa1 和 sa2 都是 shell 脚本，也是系统管理员学习 bash 编程的经典教例。

2. 查看过去的系统活动信息

sar 的基本使用方法大家早已熟练，比如使用 -d 参数查看磁盘 I/O，-r 查看系统内存状态等，本节不再重复，具体参数可以详细阅读 man 手册。本节重点讲解查找过去某一时段内系统状态的方法，比如需要查找过去某个时刻系统进程数最高的那个时间点与进程数时，可以读取位于 /var/log/sa 中历史的 sar 数据，然后利用 sort 命令对指定列进行排序，示例代码如下：

```
[root@server ~]# sar -f /var/log/sa/sa22 -q | sort -nr -k 3| more
Average:          17       878      3.15      4.56      4.93
02:30:01 PM       56      1044      4.20      5.90      6.04
01:20:01 AM       51       908      4.29      5.26      5.67
05:30:01 PM       50       911      2.53      3.03      3.41
08:30:01 AM       46       931      3.48      5.59      8.40
06:40:01 AM       46       913      4.24      5.40      5.72
08:20:01 AM       44       920      4.20      7.34     10.78
08:10:01 AM       44       999      5.50     19.59     15.37
04:30:01 AM       44       919      1.89      2.97      3.24
......

Linux 2.6.32-279.5.2.el6.x86_64  (server)   05/22/2015   _x86_64_   (24 CPU)
12:00:01 AM   runq-sz  plist-sz   ldavg-1   ldavg-5  ldavg-15
```

sar -q 用于显示队列与进程数，从文件中读取的时候，只需要使用 –f 参数指定对应的 sar 信息文件即可。

第三列 plist-sz 是系统的进程数，使用 sort 对第三列进行排列就可以找出最大进程数与最大进程数的时间。

2.3 软件分析的基本工具

2.3.1 ldd

ldd 是一个用来查看程序运行所需共享库的工具。它会告诉用户这个程序依赖了哪些库文件、库文件的位置，以及是否缺少库文件等。

ldd 其实只是一个 shell 的脚本，其原理是调用 ld-linux.so 模块来查看程序依赖的共享库。示例代码如下：

```
[root@el6-build ~]# ldd /usr/bin/mysql
    linux-vdso.so.1 =>  (0x00007fff4d7df000)
    libncursesw.so.5 => /lib64/libncursesw.so.5 (0x00007f21347eb000)
    libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f21345cd000)
    libmysqlclient.so.16 => /usr/lib64/mysql/libmysqlclient.so.16 (0x00007f
        2134249000)
    libcrypt.so.1 => /lib64/libcrypt.so.1 (0x00007f2134012000)
    libnsl.so.1 => /lib64/libnsl.so.1 (0x00007f2133df8000)
    libssl.so.10 => /usr/lib64/libssl.so.10 (0x00007f2133b8d000)
    libcrypto.so.10 => /usr/lib64/libcrypto.so.10 (0x00007f21337ae000)
    libz.so.1 => /lib64/libz.so.1 (0x00007f2133597000)
    libstdc++.so.6 => /usr/lib64/libstdc++.so.6 (0x00007f2133291000)
    libm.so.6 => /lib64/libm.so.6 (0x00007f213300d000)
    libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007f2132df6000)
    libc.so.6 => /lib64/libc.so.6 (0x00007f2132a62000)
    libtinfo.so.5 => /lib64/libtinfo.so.5 (0x00007f2132841000)
    libdl.so.2 => /lib64/libdl.so.2 (0x00007f213263c000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f2134a24000)
    libfreebl3.so => /lib64/libfreebl3.so (0x00007f21323c5000)
    libgssapi_krb5.so.2 => /lib64/libgssapi_krb5.so.2 (0x00007f2132181000)
    libkrb5.so.3 => /lib64/libkrb5.so.3 (0x00007f2131e9a000)
    libcom_err.so.2 => /lib64/libcom_err.so.2 (0x00007f2131c96000)
    libk5crypto.so.3 => /lib64/libk5crypto.so.3 (0x00007f2131a6a000)
    libkrb5support.so.0 => /lib64/libkrb5support.so.0 (0x00007f213185e000)
    libkeyutils.so.1 => /lib64/libkeyutils.so.1 (0x00007f213165b000)
    libresolv.so.2 => /lib64/libresolv.so.2 (0x00007f2131441000)
    libselinux.so.1 => /lib64/libselinux.so.1 (0x00007f2131221000)
```

2.3.2 strace 与 ltrace

在 Linux 系统中，系统调用（system call）就是内核态给用户态提供的一个系统接口，通过这个接口，可以非常容易地从用户态切换到内核态工作，strace 和 ltrace 就是用于追踪这种系统调用的，strace 与 ltrace 分别用来跟踪进程的系统调用和库函数调用。

下面用一个非常简单的 python 脚本来演示下如何使用 strace 与 ltrace。

首先创建一个 python 脚本，只需要打印 hello world 即可。然后使用这个脚本作为 strace 和 ltrace 的示例。

```
[root@server ~]# cat hello.py
#!/usr/bin/python
print 'hello world!'
```

1. strace

（1）查看 hello.py 脚本运行过程中系统调用的全过程

查看脚本运行时系统调用的命令非常简单，示例代码如下，从输出中可以看到，在执行这个 python 脚本的过程中，系统在背后做了很多的事情，因为输出太长，这里只选取开头部分。

```
[root@server ~]# strace ./hello.py
execve("./hello.py", ["./hello.py"], [/* 22 vars */]) = 0
brk(0)                                  = 0x1a46000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
    0x7f2ef2379000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=69138, ...}) = 0
mmap(NULL, 69138, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f2ef2368000
close(3)                                = 0
open("/lib64/libpython2.7.so.1.0", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0p\363\3\0\0\0\0\0"..., 832)
    = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1822488, ...}) = 0
mmap(NULL, 3954184, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
    0x7f2ef1d94000
mprotect(0x7f2ef1f0c000, 2097152, PROT_NONE) = 0
mmap(0x7f2ef210c000, 258048, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
    DENYWRITE, 3, 0x178000) = 0x7f2ef210c000
mmap(0x7f2ef214b000, 58888, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
    ANONYMOUS, -1, 0) = 0x7f2ef214b000
close(3)                                = 0
open("/lib64/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\2401\0\0\0\0\0\0"..., 832)
    = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=141616, ...}) = 0
mmap(NULL, 2208864, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
    0x7f2ef1b78000
mprotect(0x7f2ef1b8e000, 2097152, PROT_NONE) = 0
mmap(0x7f2ef1d8e000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
    DENYWRITE, 3, 0x16000) = 0x7f2ef1d8e000
mmap(0x7f2ef1d90000, 13408, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
    ANONYMOUS, -1, 0) = 0x7f2ef1d90000
close(3)                                = 0
open("/lib64/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\320\16\0\0\0\0\0\0"..., 832)
    = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=19512, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
```

```
    0x7f2ef2367000
mmap(NULL, 2109744, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
    0x7f2ef1974000
mprotect(0x7f2ef1977000, 2093056, PROT_NONE) = 0
```

比如 open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3 这行，open 的这种系统调用可以通过 man 手册来查阅，具体可以查阅 man syscalls。

（2）统计有多少个系统调用

使用 -c 参数可以统计出所有的系统调用与调用次数。示例代码如下：

```
[root@server ~]# strace -c ./hello.py
hello world!
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 16.31    0.001752           9       186       122 open
 15.36    0.001650          28        59           mmap
 13.75    0.001477          22        68           rt_sigaction
 10.58    0.001136          81        14           mprotect
  9.51    0.001021          10       106           read
  7.76    0.000833          13        66           close
  7.32    0.000786           8        99           fstat
  6.01    0.000645           7        89        61 stat
  3.67    0.000394          12        32           munmap
  3.00    0.000322          11        30           brk
  1.31    0.000141         141         1           execve
  0.84    0.000090          18         5         1 ioctl
  0.74    0.000080          80         1         1 access
  0.74    0.000080          80         1           set_tid_address
  0.72    0.000077          77         1           set_robust_list
  0.66    0.000071          71         1           arch_prctl
  0.39    0.000042          21         2           openat
  0.34    0.000037          37         1           rt_sigprocmask
  0.34    0.000037          37         1           getrlimit
  0.34    0.000036           6         6           lstat
  0.32    0.000034          11         3           lseek
  0.00    0.000000           0         1           write
  0.00    0.000000           0         4           getdents
  0.00    0.000000           0         1           getcwd
  0.00    0.000000           0         6         2 readlink
  0.00    0.000000           0         1           getuid
  0.00    0.000000           0         1           getgid
  0.00    0.000000           0         1           geteuid
  0.00    0.000000           0         1           getegid
------ ----------- ----------- --------- --------- ----------------
100.00    0.010741                   788       187 total
```

（3）按照 calls 的次数排序

如果希望知道 syscall 中哪几种 call 最多，可以使用如下代码：

```
[root@server ~]# strace -c -S calls ./hello.py
```

```
hello world!
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 10.62    0.000684           4       186       122 open
  6.97    0.000449           4       106           read
  8.18    0.000527           5        99           fstat
  0.57    0.000037           0        89        61 stat
  3.74    0.000241           4        68           rt_sigaction
  9.56    0.000616           9        66           close
 28.12    0.001811          31        59           mmap
  3.03    0.000195           6        32           munmap
  1.29    0.000083           3        30           brk
 18.12    0.001167          83        14           mprotect
  0.00    0.000000           0         6           lstat
  0.00    0.000000           0         6         2 readlink
  0.00    0.000000           0         5         1 ioctl
  0.00    0.000000           0         4           getdents
  0.00    0.000000           0         3           lseek
  0.00    0.000000           0         2           openat
  0.00    0.000000           0         1           write
  1.34    0.000086          86         1           rt_sigprocmask
  1.47    0.000095          95         1         1 access
  1.49    0.000096          96         1           execve
  0.00    0.000000           0         1           getcwd
  1.34    0.000086          86         1           getrlimit
  0.00    0.000000           0         1           getuid
  0.00    0.000000           0         1           getgid
  0.00    0.000000           0         1           geteuid
  0.00    0.000000           0         1           getegid
  1.57    0.000101         101         1           arch_prctl
  1.32    0.000085          85         1           set_tid_address
  1.27    0.000082          82         1           set_robust_list
------ ----------- ----------- --------- --------- ----------------
100.00    0.006441                   788       187 total
```

（4）只看某一种 syscall 的调用情况

下面的代码会使用 -e 参数指定系统调用的类型。

```
[root@server ~]# strace -c -e open ./hello.py
hello world!
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
100.00    0.002185          12       186       122 open
------ ----------- ----------- --------- --------- ----------------
100.00    0.002185                   186       122 total
```

2. ltrace

ltrace 的用法与 strace 类似，重点在函数调用方面。

（1）跟踪库函数的调用

在 ltrace 里跟踪库函数的调用可使用 -cf 参数，示例代码如下：

```
[root@server ~]# ltrace -cf grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
   % time     seconds  usecs/call       calls      function
------      ----------- -----------   ---------  --------------------
    19.50    0.006624          86        77 malloc
    18.01    0.006116          98        62 strlen
    12.34    0.004190          59        71 free
     8.40    0.002853          95        30 realloc
     7.83    0.002660          80        33 __ctype_get_mb_cur_max
     7.74    0.002629          97        27 strcpy
     7.05    0.002395          95        25 strncmp
     6.54    0.002220          69        32 mbrtowc
     1.65    0.000562          70         8 calloc
     1.14    0.000386         193         2 setlocale
     1.12    0.000380          95         4 wcrtomb
     0.88    0.000298          74         4 wctob
     0.82    0.000278         139         2 read
     0.72    0.000246         246         1 re_compile_pattern
     0.65    0.000221          55         4 memchr
     0.46    0.000157         157         1 __cxa_atexit
     0.44    0.000148          74         2 __fpending
     0.43    0.000145          72         2 fwrite_unlocked
     0.37    0.000124         124         1 getpagesize
     0.36    0.000123         123         1
     0.36    0.000122          61         2 fclose
     0.34    0.000117          58         2 isatty
     0.33    0.000111         111         1 strrchr
     0.32    0.000109         109         1 _obstack_begin
     0.32    0.000107         107         1 close
     0.25    0.000084          84         1 bindtextdomain
     0.24    0.000080          80         1 textdomain
     0.24    0.000080          80         1 getopt_long
     0.23    0.000079          79         1 re_set_syntax
     0.23    0.000078          78         1 strcmp
     0.22    0.000076          76         1 getenv
     0.17    0.000059          59         1 __xstat
     0.17    0.000058          58         1 open
     0.15    0.000050          50         1 nl_langinfo
------      -----------    -----------   ---------  --------------------
   100.00    0.033965                    405 total
```

（2）追踪一个进程的库函数调用

这里以服务器上的一个 mysql 进程为例，首先获取到 mysql 进程的 pid，然后在 ltrace 中使用 -p 参数加上 mysql 的 pid 即可追踪 mysql 这个进程的库函数调用。

```
[root@server ~]# ps -aux | grep mysql
root        1704  0.0  0.0 106064  1488 pts/0    S    02:04   0:00 /bin/sh /usr/
    bin/mysqld_safe  --datadir=/var/lib/mysql  --socket=/var/lib/mysql/mysql.sock
    --pid-file=/var/run/mysqld/mysqld.pid --basedir=/usr --user=mysql
```

```
mysql      1806  0.0  0.6 367948 27288 pts/0    Sl   02:04   0:05 /usr/libexec/
    mysqld --basedir=/usr --datadir=/var/lib/mysql --user=mysql --log-error=/
    var/log/mysqld.log --pid-file=/var/run/mysqld/mysqld.pid --socket=/var/lib/
    mysql/mysql.sock
root      29285  0.0  0.0 103312   824 pts/0    S+   11:53   0:00 grep mysql

[root@server ~]# ltrace -p 1806
[pid 1813] pthread_mutex_trylock(0x7fe5bff02920, 0, 0, -1, 0x7fe5bcc3dd30) = 0
[pid 1813] pthread_mutex_unlock(0x7fe5bff02920, 0, 0, 0x7fe5c81af628,
    0x7fe5bcc3dd30) = 0
[pid 1813] time(NULL) = 1432871609
[pid 1813] difftime(0x5567e2b9, 0x5567e28b, 859093, 0x20c49ba5e353f7cf,
    0x963e07f8e9ca) = 0x5567e2b9
[pid 1813] pthread_mutex_lock(0x1669070, 0x5567e28b, 859093, 0x20c49ba5e353f7cf,
    0x963e07f8e9ca) = 0
[pid 1813] pthread_mutex_unlock(0x1669070, 3, 1, 0x20c49ba5e353f7cf, 0x1669070) = 0
[pid 1813] pthread_mutex_lock(0x1669070, 0x7fe5bcc3dd90,0x1669070,
    0x20c49ba5e353f7cf, 0x1669070) = 0
[pid 1813] pthread_mutex_unlock(0x1669070, 9999, 1, 0x20c49ba5e353f7cf,
    0x1669070) = 0
[pid 1813] fflush(0x7fe5c6b4c860) = 0
[pid 1813] select(0, 0, 0, 0, 0x7fe5bcc3dd30 <unfinished ...>
[pid 1812] pthread_mutex_trylock(0x7fe5bfeff2c8, 0, 0, -1, 0x7fe5bd63ed70) = 0
[pid 1812] pthread_mutex_lock(0x19864b0, 0, 0, 0x7fe5c81af628, 0x7fe5bd63ed70) = 0
[pid 1812] pthread_mutex_unlock(0x19864b0, 3, 1, 0x7fe5c81af628, 0x19864b0) = 0
[pid 1812] pthread_mutex_unlock(0x7fe5bfeff2c8, 0, 0x19864b0, 0x7fe5c81af628,
    0x19864b0) = 0
[pid 1812] select(0, 0, 0, 0, 0x7fe5bd63ed70 <unfinished ...>
[pid 1813] <... select resumed> ) = 0
[pid 1813] pthread_mutex_trylock(0x7fe5bff02920, 0, 0, -1, 0x7fe5bcc3dd30) = 0
[pid 1813] pthread_mutex_unlock(0x7fe5bff02920, 0, 0, 0x7fe5c81af628,
    0x7fe5bcc3dd30)
......
```

2.3.3　ipcs

进程间通信是系统中常见的场景，多个进程可能会需要调用同一个内存内容，比如管道，前一个进程的输出放入内存，后一个命令去读取这段内存。

一共有三种进程间通信方法：

- ❑ semaphores：表示信号量。
- ❑ message queues：表示消息队列。
- ❑ share memory regions：表示共享内存段。

用户可以使用 ipcs 这个命令来查看以上三种进程间通信的具体情况，示例如下：

```
[root@server ~]# ipcs

------ Message Queues --------
```

```
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x01125aae 0          root       600        1000       9

------  Semaphore Arrays --------
key          semid      owner     perms      nsems
0x00000000 131072     apache     600        1
0x00000000 163841     apache     600        1
0x00000000 196610     apache     600        1
0x00000000 229379     apache     600        1
0x00000000 262148     apache     600        1
```

1. 配置共享内存

一般情况下，系统管理员很少遇到处理共享内存的情况，系统默认的配置已经足够使用，所以这里只做简单的讲解。

假设现在因为一些需求，要限制进程申请的共享内存空间最大 1024MB。

首先，使用 ipcs -l -m 查看到现在系统的最大共享内存空间，这里为 1073741824，在 kernel 中这个参数是由 kernel.shmall 控制的，kernel.shmall 的单位是 page，所以要将 1024MB 转换为 page 数目。使用 sysctl -w 可让修改即时生效。示例如下：

```
[root@server ~]# ipcs -l -m
------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 4194303
max total shared memory (kbytes) = 1073741824
min seg size (bytes) = 1

[root@server ~]# echo $[1024*1024/4]
262144
[root@server ~]# sysctl -w kernel.shmall=262144
kernel.shmall = 262144
[root@server ~]# ipcs -l -m

------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 4194303
max total shared memory (kbytes) = 1048576
min seg size (bytes) = 1
```

2. 清除共享内存

清除共享内存也是一个很少会触发的动作，但还是要知道如何使用 ipcs 命令查看现在的共享内存段，可使用 ipcrm 清除共享内存。

示例如下：

```
[root@server ~]# ipcs -m
```

```
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x0112536f 0          root       600        1000       6

[root@server ~]# ipcrm -M 0x0112536f
[root@server ~]# ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 0          root       600        1000       6          dest
```

2.3.4　systemtap

systemtap 是一个非常著名的内核态进程跟踪程序，它的作用非常广泛，最主要的作用是寻找程序的性能瓶颈。

Linux 内核里有一个 kprobe 机制，这是一个动态的收集 debug 信息的工具，systemtap 就是基于 kprobe 机制的一个调试工具。systemtap 的使用简单，而且可以自己定义脚本，对开发人员、系统管理员来说是一个深入分析性能的利器。本节将讲述如何安装配置 systemtap，以及如何使用已有的 systemtap 脚本。

1. 安装准备

systemtap 的安装需要准备一台测试机器，先在测试机器上安装 kernel-debuginfo、kernel-debuginfo-common、kernel-level、systemtap-runtime、gcc 等相关包，然后在测试机器上编译测试脚本，编译测试完成之后将编译好的模块放在生产机器上，生产机器只需要安装 systemtap-runtime 包即可。

CentOS 的 debuginfo 相关安装包可以在 http://debuginfo.centos.org/ 里找到。

注意，安装 kernel-debuginfo 时，相应的内核版本一定要一致！

2. 安装 kernel 相关包

首先要确定系统内核版本，然后下载相对应的 kernel-debuginfo 包。示例如下：

```
[root@systemtap ~]# uname -a
Linux systemtap.example.com 2.6.32-504.el6.x86_64 #1 SMP Wed Oct 15 04:27:16 UTC
    2014 x86_64 x86_64 x86_64 GNU/Linux

[root@systemtap ~]# wget http://debuginfo.centos.org/6/x86_64/kernel-debug-
    debuginfo-2.6.32-504.el6.x86_64.rpm

[root@systemtap ~]# wget http://debuginfo.centos.org/6/x86_64/kernel-debuginfo-
    2.6.32-504.el6.x86_64.rpm

[root@systemtap ~]# wget http://debuginfo.centos.org/6/x86_64/kernel-debuginfo-
    common-x86_64-2.6.32-504.el6.x86_64.rpm
```

```
[root@systemtap ~]# rpm -ivh kernel-debuginfo-common-x86_64-2.6.32-504.el6.
    x86_64.rpm
Preparing...                ########################################### [100%]
    1:kernel-debuginfo-common########################################### [100%]
[root@systemtap ~]# rpm -ivh kernel-debuginfo-2.6.32-504.el6.x86_64.rpm
Preparing...                ########################################### [100%]
    1:kernel-debuginfo      ########################################### [100%]
[root@systemtap ~]# rpm -ivh kernel-debug-debuginfo-2.6.32-504.el6.x86_64.rpm
Preparing...                ########################################### [100%]
    1:kernel-debug-debuginfo ########################################### [100%]
```

3. 安装 Systemtap 软件包

安装命令如下：

```
[root@systemtap ~]# yum -y install install gcc systemtap systemtap-runtime
    kernel-debug kernel-devel kernel-debug-devel kernel-firmware
```

4. 编译 Systemtap 脚本

Systemtap 软件包本身已经提供了一些脚本，这些脚本涉及 I/O、内存、网络等，并且几乎可以直接使用。下面以 profiling 中的 topsys.stp 为例，讲解如何编译。

```
[root@systemtap ~]# cd /usr/share/doc/systemtap-client-2.5/example
[root@systemtap examples]# ls -lrt
total 412
-rw-r--r-- 1 root root   5886 Oct 15  2014 README
-rw-r--r-- 1 root root  91092 Oct 15  2014 keyword-index.txt
-rw-r--r-- 1 root root 140946 Oct 15  2014 keyword-index.html
-rw-r--r-- 1 root root  47261 Oct 15  2014 index.txt
-rw-r--r-- 1 root root  79298 Oct 15  2014 index.html
drwxr-xr-x 3 root root   4096 Jun  1 00:29 general
drwxr-xr-x 2 root root   4096 Jun  1 00:29 html
drwxr-xr-x 2 root root   4096 Jun  1 00:29 interrupt
drwxr-xr-x 2 root root   4096 Jun  1 00:29 io
drwxr-xr-x 2 root root   4096 Jun  1 00:29 locks
drwxr-xr-x 2 root root   4096 Jun  1 00:29 memory
drwxr-xr-x 2 root root   4096 Jun  1 00:29 network
drwxr-xr-x 2 root root   4096 Jun  1 00:29 process
drwxr-xr-x 2 root root   4096 Jun  1 00:29 profiling
drwxr-xr-x 3 root root   4096 Jun  1 00:29 stapgames
drwxr-xr-x 2 root root   4096 Jun  1 00:29 virtualization
```

这里将 profiling 中的 topsys.stp 拷贝到 /tmp 目录，然后查看这个文件。

这段脚本的作用是每隔 5 秒钟列出当前系统中最高的 20 个 systemcalls。这个脚本对一些系统软件的开发人员作用较大。脚本如下：

```
#!/usr/bin/stap
#
# This script continuously lists the top 20 systemcalls in the interval
# 5 seconds
#
```

```
global syscalls_count

probe syscall.* {
    syscalls_count[name] <<< 1
}

function print_systop () {
    printf ("%25s %10s\n", "SYSCALL", "COUNT")
    foreach (syscall in syscalls_count- limit 20) {
        printf("%25s %10d\n", syscall, @count(syscalls_count[syscall]))
    }
    delete syscalls_count
}

probe timer.s(5) {
    print_systop ()
printf("------------------------------------------------------------\n")
}
```

下面使用 -v 参数试跑这个脚本，当输出为 pass 5 ：starting run 时，就证明这个脚本通过了调试，可以编译成模块了。

```
[root@systemtap tmp]# stap -v topsys.stp
Pass 1: parsed user script and 103 library script(s) using 201636virt/29536res/3
    156shr/26856data kb, in 120usr/10sys/142real ms.
Pass 2: analyzed script: 429 probe(s), 44 function(s), 41 embed(s), 1 global(s)
    using 306308virt/135184res/4204shr/131528data kb, in 960usr/230sys/1863real ms.
Pass 3: translated to C into "/tmp/stap5s2AT2/stap_6421353660f490975a2a346ed9
    c7ed0f_182267_src.c" using 306308virt/135576res/4596shr/131528data kb, in
    40usr/30sys/79real ms.
Pass 4: compiled C into "stap_6421353660f490975a2a346ed9c7ed0f_182267.ko" in
    6980usr/760sys/9346real ms.
Pass 5: starting run.
                  SYSCALL      COUNT
                     read         27
                    ppoll         25
                    fcntl          4
                  pselect6         1
------------------------------------------------------------
```

通过测试之后，使用 -p4 -m < 模块名 > < 脚本 > 的命令行即可将脚本编译成一个模块，如下：

```
[root@systemtap tmp]# stap -v -p4 -m topsys.ko topsys.stp
Truncating module name to 'topsys'
Pass 1: parsed user script and 103 library script(s) using 201408virt/29580res/3
    200shr/26628data kb, in 120usr/10sys/129real ms.
Pass 2: analyzed script: 429 probe(s), 44 function(s), 41 embed(s), 1 global(s)
    using 306284virt/135196res/4224shr/131504data kb, in 1010usr/80sys/1090real ms.
Pass 3: translated to C into "/tmp/stapX0bYk3/topsys_src.c" using 306284virt/135
```

```
    516res/4544shr/131504data kb, in 40usr/40sys/77real ms.
topsys.ko
Pass 4: compiled C into "topsys.ko" in 3900usr/180sys/4322real ms.
[root@systemtap tmp]# ll topsys.ko
-rw-r--r-- 1 root root 657770 Jun  1 00:34 topsys.ko
```

5. 测试模块

测试模块时，先在另外一台机器上安装 systemtap-runtime 包，然后使用 staprun 直接 run 编译好的模块。此时就可以看到有多少的 systemcall 了。

下面用 cat 命令向 /dev/null 重定向写入，同时打开新的终端，运行 staprun topsys.ko 去查看 systemcall 的实时情况。

```
[root@test ~]# staprun topsys.ko
               SYSCALL     COUNT
                  read        27
                 ppoll        25
                 fcntl         4
              pselect6         1
-----------------------------------------------------------

[root@test ~]# cat /dev/zero > /dev/null

[root@test ~]# staprun topsys.ko
               SYSCALL     COUNT
                  read   1914489
                 write   1914462
                 ppoll        25
                 fcntl         4
              pselect6         1
                  poll         1
-----------------------------------------------------------
               SYSCALL     COUNT
                  read   1912028
                 write   1912003
                 ppoll        25
       rt_sigprocmask         4
               select         2
-----------------------------------------------------------
```

可以看到，此时系统出现了大量的 read 和 write 的 syscall，两者相差数量不大，这也是比较典型文件复制时会出现的场景。

2.4 与内存相关的那些事情

2.4.1 内存泄漏

如果程序在运行过程中不能正常回收不用的内存，那么时间一长就会导致内存增长很

高，最终导致系统不可用，这种情况叫做内存泄漏。内存泄漏是一个让人烦恼的问题，不仅系统管理员烦恼，程序员也烦恼这个问题，所以定位分析内存泄漏是每一个系统管理员需要掌握的技能。

开源的 valgrind 是一个非常易于上手的内存分析工具，它可以分析内存泄漏、缓存命中等。本节以笔者工作中曾遇到的一个问题为例，讲述如何使用 valgrind 来定位内存泄漏问题。

笔者在生产环境使用 Puppet 作为自动化的基础工具，但是发现在 CentOS 5 的系统中，Puppet 进程一段时间之后使用了太多的系统内存，导致服务器宕机。根据监控系统可以看到 Puppet 进程的内存存在缓慢持续增长的趋势，因此怀疑 Puppet 有内存泄漏的问题，于是开始查找证据。

首先，用 valgrind 来分析 Puppet 在运行过程中是否出现内存泄漏问题。在 leak summary 中，很明确地指出了内存泄漏，泄漏了多少的量，如下：

```
[root@server1 ~]# valgrind --tool=memcheck /usr/sbin/puppetd -t
==1794== Memcheck, a memory error detector
==1794== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==1794== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==1794== Command: /usr/sbin/puppetd -t
==1794==
==1794== Conditional jump or move depends on uninitialised value(s)
......
==1794==
==1794== HEAP SUMMARY:
==1794==     in use at exit: 23,430,539 bytes in 136,643 blocks
==1794==   total heap usage: 702,272 allocs, 565,629 frees, 262,185,929 bytes
    allocated
==1794==
==1794== LEAK SUMMARY:
==1794==    definitely lost: 617 bytes in 18 blocks
==1794==    indirectly lost: 0 bytes in 0 blocks
==1794==      possibly lost: 84,736 bytes in 1,523 blocks
==1794==    still reachable: 23,345,186 bytes in 135,102 blocks
==1794==         suppressed: 0 bytes in 0 blocks
==1794== Rerun with --leak-check=full to see details of leaked memory
==1794==
==1794== For counts of detected and suppressed errors, rerun with: -v
==1794== Use --track-origins=yes to see where uninitialised values come from
==1794== ERROR SUMMARY: 583211 errors from 100 contexts (suppressed: 19 from 6)
```

因为 Puppet 是由 Ruby 所写，所以内存泄漏的原因可能是来自 Ruby 本身。查看系统，得知安装的 Ruby 版本为 1.8.5 版本。在将 Ruby 升级到 1.8.7 版本之后，发现内存泄漏情况缓解了很多，不过依然有泄漏情况存在。示例如下：

```
[root@server1 ~]# rpm -qa | grep ruby
ruby-libs-1.8.5-31.el5_9
```

```
ruby-shadow-1.4.1-8.el5
ruby-1.8.5-31.el5_9
ruby-augeas-0.4.1-2.el5
```

可以看到，此时尽管内存泄露的情况缓解了很多，但是无法阻止问题再次发生，于是设置了一个 cronjob 定期重启 Puppet 进程，以保证避免内存泄漏而导致服务器宕机的问题发生。

2.4.2 虚拟内存、物理内存与页缺失

众所周知，在计算机发展的开始，内存是一个稀缺资源，即便现在的服务器动辄 128GB 的内存，它依然是一个稀缺的系统资源。所以内存管理也是影响性能的因素之一。

对于内存，首先，要清楚内存的单位，Paging 是内存的最小单位，称为页，类似磁盘的 block 概念，默认情况一个页是 4KB 大小。

通常情况下，对于内存的分配，并不是进程申请多少内存，操作系统就给多少内存。一般来说，当进程向操作系统申请 10GB 的内存时，操作系统收到请求后会进行自我检查，经过分析决定给予进程 10GB 的内存空间，此时操作系统会对这个进程说：“Hi，我可以给你 10GB 的内存空间。”这样的内存称为进程虚拟内存。而此时，操作系统并没有真正给予进程 10GB 内存，操作系统还会对进程说：“Hi，虽然我可以给你 10GB 内存空间，但现在你实际只需要 300MB 的内存就可以运行程序了，所以物理内存中现在只划分了 300MB 给你。”这种实际分配给进程的内存叫做物理内存。

在系统中可以使用 ps 查看进程的虚拟内存与物理内存大小，如下：

```
[root@server ~]# ps aux | head -1
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
```

其中，VSZ 这列代表的是虚拟内存，RSS 这列代表的是物理内存。

由于进程不能直接获取物理内存，而是每一个进程都有一个虚拟内存空间，所以虚拟内存不会直接映射到物理内存，直到使用的时候才会出现映射关系。这里就会产生一个问题，当进程向操作系统请求内存时，可能操作系统并没有准备好。这种情况叫做内存页缺失，页缺失有两种情况：一种是主页缺失，一种是次页缺失。

（1）主页缺失

如果进程请求的数据不在物理内存中，要从磁盘或者交换分区换到内存中，这种叫做主页缺失，这种情况是非常影响性能的。

（2）次页缺失

当第一次物理内存被使用的时候，物理内存中其实没有分配，这时候就产生了一个次页缺失。次页缺失对性能的影响不会特别大，一般情况下可以忽略。

下面说明如何查看进程的页缺失情况，笔者截取了生产系统中一台 KVM 主机的情况，如下：

```
[root@kvm01 ~]# ps o pid,comm,minflt,majflt `pidof qemu-kvm`
  PID COMMAND         MINFLT MAJFLT
 2834 qemu-kvm        443408224 117369
 2948 qemu-kvm        224733987 63896
19651 qemu-kvm        270213039 66725
20011 qemu-kvm        123041546 66718
20862 qemu-kvm        550133562 197954
26869 qemu-kvm        632148562 165397
32601 qemu-kvm        1103935202 165473
```

其中，MINFLT 是次页缺失，MAJFLT 是主页缺失。可以看到虚拟机比较繁忙，有大量的主页和次页缺失。

2.4.3　Out of Memory

Out of Memory（OOM）是系统管理员的另外一个噩梦。OOM 发生的原因主要在于，当发生次页缺失的时候，恰好系统无法再释放出物理内存，此时系统只有杀掉一些进程来释放物理内存。

OOM 会选择占用内存最多的那个进程开始杀进程，一直到内存足够为止。但是这样的方式往往会造成一些困扰，因为关键进程被 kill 掉，相当于宕机。其实可以让 OOM 不杀进程，而是让 kernel panic，可通过如下方式来实现。

设置 /proc/sys/vm/panic_on_oom 为 1，这样在发生 OOM 的时候就会让 kernel panic，示例如下：

```
[root@kvm01 ~]# cat /proc/sys/vm/panic_on_oom
0
[root@kvm01 ~]# echo 1 > /proc/sys/vm/panic_on_oom
```

实际情况中，如果 OOM killer 发生了，说明内存真的不足了。这时需要重视内存的使用情况，评估是否需要增加内存。

2.4.4　Overcommit

前面说到了虚拟内存与物理内存的关系，一般情况下进程并不会一次用光申请的内存，所以操作系统为了提高内存使用率，会向进程“超卖”内存，以便能响应更多的进程内存申请，当然，操作系统有时候也是有点节操的，它并不会无限制响应进程的内存申请，这可防止内存申请过多，导致操作系统本身的内存空间不足而无法正常运行。Linux 系统中使用 Overcommit 的方式来控制内存的申请。

控制是否允许内存“超卖”是通过 /proc/sys/vm/overcommit_memory 来实现的，它有三种模式，分别是 0、1、2。示例如下：

```
[root@kvm01 ~]# cat /proc/sys/vm/overcommit_memory
0
```

- 0 是系统默认的模式，在这种模式下，系统会尽可能地响应进程的内存申请，这样一来，有可能发生前面一节所说的 Out of Memory 情况。
- 1 的模式下，系统完全响应进程的内存申请，不管自己的资源还剩下多少。
- 2 的模式下，系统完全不允许进程申请超过系统设置大小的内存空间。在这种模式下，系统设置的可申请内存空间大小是：

Swap+RAM*（“/proc/sys/vm/overcommit_ratio”/100）

其中，/proc/sys/vm/overcommit_ratio 就是系统最大可分配内存的百分比，默认为 50，也就是说最大为 50%。

在使用 2 模式时，在 /proc/meminfo 中 CommitLimit 和 Committed_AS 这两个值会显得比较重要，CommitLimit 就是系统可以申请虚拟内存的最大值，Committed_AS 是系统已经申请的虚拟内存的大小。

2.4.5 cache 与 buffer

cache 与 buffer 这两个词都有缓存的意思，故而成为大家争论的焦点，而且在数据库中也有这两个词，所以网络上众说纷纭，这里只说在 free 命令中出现的 cache 与 buffer。

buffer 指的是索引信息，就是对磁盘文件的索引缓存，这个值一般比较低。

cache 则是传统意义上的缓存，它缓存的是文件内容。

关于 buffer 和 cache 使用的计算方式请查看前面 2.2.2 节中 free 的用法。这里也给出一个示例，如下：

```
[root@ash0007 ~]# free -m
             total       used       free     shared    buffers     cached
Mem:         23985      22409       1576          0         14        352
-/+ buffers/cache:      22041       1944
Swap:        20294       1942      18352
```

2.5 与磁盘相关的那些事情

2.5.1 HDD 与 SSD

硬件的发展真的非常快，从 SSD 固态硬盘出现到大规模进入各个公司的生产环境仅仅几年时间，同时 HDD 机械硬盘的发展也未停滞，HDD 硬盘虽然读写速度没有质的突破，但是容量却在不断攀升，现在市场上已经可以买到 8TB 的单块硬盘，所以在未来的很长的一段时间里，HDD 和 SSD 会在服务器市场上并存。

（1）HDD

HDD 性能瓶颈主要两个方面：seek time 寻址时间与 rotational delay 旋转延时。

HDD 磁盘读数据的过程是先找到磁道，然后找磁道上的扇区。找到磁道的时间叫做寻址时间，也就是 seek time；找到扇区的时间叫做旋转延时，也就是 rotational delay，这两

个动作的时间加起来可能会达到 1s 以上。所以针对 seek time 和 rotation delay 做优化时主要是减少寻找的时间，以及寻找的次数。

（2）SSD

SSD 发展迅速，变化也较快，接口包括 SATA、mSATA、PCI-E 等，存储的颗粒包括 MLC、SLC、TLC 等。

- MLC：一个存储单元存储多个（通常是两个）比特位的信息，寿命比 SLC 短，读写速度慢于 SLC，但是价格便宜。
- SLC：一个存储单元只存储一个比特位的信息，寿命长，读写速度快，价格昂贵。
- TLC ：一个存储单元存储多个（通常是三个）比特位的信息，速度慢，寿命短，但是价格便宜，多用于手机存储芯片。

2.5.2　HDD 磁盘的调度算法

HDD 的速度较慢（这不是绝对的！），针对机械硬盘的特性，解决该问题有两种方式：

1）加入中间缓存，比如增大 HDD 上的缓存。

2）对于 I/O 请求合并操作，尽可能顺序写入，顺序读取。

根据这两种方式，操作系统会针对不同的应用场景采用不同的磁盘调度方式，查看系统配置的磁盘调度算法如下：

```
[root@server ~]# cat /sys/block/sda/queue/scheduler
noop anticipatory [deadline] cfq
```

系统提供了四种调度算法，CentOS 6 默认设置在 deadline 中，下面讲讲这四种调度算法的区别。

- noop（无操作等待算法）：不干预任何的 I/O 请求，直接将 I/O 请求交给存储设备，由存储设备自己完成。这个常出现在使用 SAN 的场景下，由存储自己完成 I/O 合并优化，或者出现在虚拟机上。宿主机自己会完成 I/O 请求的合并，虚拟机不需要做 I/O 请求合并。
- anticipatory（预期算法）：预期调度会将 I/O 请求放进队列，但并不立刻完成，而是在合并成顺序 I/O 后再完成请求，对于持续大量顺序 I/O 的场景适合使用 anticipatory。
- deadline（最后期限）：将 I/O 请求放进队列不处理，一直等到队列中的 I/O 请求多到足够合并成一个比较好的 I/O 请求为止。这个方式适合虚拟化环境的物理机，数据库服务器。
- cfq（完全公平队列）：对每一个进程的 I/O 请求公平处理，I/O 响应很快，适合随机存取，比如文件服务器。

如果需要更改磁盘的调度算法，只需用 echo 方式将算法写入即可：

```
[root@server ~]# echo cfq > /sys/block/sda/queue/scheduler
[root@server ~]# cat /sys/block/sda/queue/scheduler
```

```
noop anticipatory deadline [cfq]
```

2.5.3 文件系统中的日志

操作系统中数据写入磁盘的方式是先写入缓存，然后再写入磁盘。如果在数据写入了缓存，还没来得及没有写入磁盘的时候，机器断电了或者宕机了，那么在机器重新启动时就会发现实际数据和预期状态不一致。为了能恢复到一致，文件系统会从磁盘划分一个日志空间，在操作系统写数据到磁盘上之前，会将脏数据优先写入日志空间，然后再同步到磁盘。

文件系统的日志写入方式有以下三种。

- ordered 方式：只记录元数据到日志空间，待元数据写入日志空间之后，再把数据写入磁盘文件系统。这种方式下文件系统的性能和数据的安全性可以做到相对的均衡。这也是大多数日志文件系统默认的方式。
- writeback 方式：元数据和数据会同时写入磁盘，这种方式提供了较好的磁盘性能，但是数据安全性无法保证。
- journal 方式：这种方式会先向日志空间写入元数据和数据，然后向文件系统再写一次元数据和数据，这种方式数据最为安全，但是因为元数据和数据都会写两份，文件系统的性能也是最差的。

2.6 系统资源限制

系统资源是有限的，有的时候为了系统安全或者为了能承载更多的压力，需要限制或放开一些进程的资源使用。在早期的 Linux 系统中，一般用 ulimit 来限制进程的资源使用，在现在的 Linux 系统中，kernel 又引入了 cgroup 进一步加强限制进程的资源的使用。

2.6.1 ulimit

ulimit 几乎可以说是所有系统管理员都必须熟练掌握的一个配置，大部分的配置都在 /etc/security/limits.conf 中，配置文件中包含了绝大部分配置的解释。比如，如何限制用户进程数、如何确定打开文件数目等，这已经是大家很熟悉的配置了，这里不多讲解，下面用 ulimit 来演示如何限制内存申请。

前面的章节讲到了虚拟内存，虚拟内存是进程向操作系统申请的内存。虚拟内存的申请也是可以用 ulimit 来限制的。

首先，使用 ulimit -a 查看现在系统中 ulimit 的设置情况，如下：

```
[root@systemtap ~]# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
```

```
scheduling priority             (-e) 0
file size                   (blocks, -f) unlimited
pending signals                 (-i) 14720
max locked memory           (kbytes, -l) 64
max memory size             (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size                (512 bytes, -p) 8
POSIX message queues         (bytes, -q) 819200
real-time priority              (-r) 0
stack size                  (kbytes, -s) 10240
cpu time                   (seconds, -t) unlimited
max user processes              (-u) 14720
virtual memory              (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

可以看到，virtual memory 一行是 unlimited，即无限的。

此时将 virtual memory 设置成 0，看看会发生什么。

```
[root@systemtap ~]# ulimit -v 0
[root@systemtap ~]# ulimit -a
core file size              (blocks, -c) 0
data seg size               (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size                   (blocks, -f) unlimited
pending signals                 (-i) 14720
max locked memory           (kbytes, -l) 64
max memory size             (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size                (512 bytes, -p) 8
POSIX message queues         (bytes, -q) 819200
real-time priority              (-r) 0
stack size                  (kbytes, -s) 10240
cpu time                   (seconds, -t) unlimited
max user processes              (-u) 14720
virtual memory              (kbytes, -v) 0
file locks                      (-x) unlimited
```

在执行 ls 命令的时候可以发现不会正确执行，而是提示 Killed！甚至连 reboot 都无法正确执行。如下：

```
[root@systemtap ~]# ls
Killed
[root@systemtap ~]# ls
Killed
[root@systemtap ~]# reboot
Killed
```

这是因为系统虚拟内存为 0，ls、reboot 无法申请到虚拟内存空间，故而这个命令无法执行，只能被系统杀掉！

2.6.2 Cgroup

ulimit 限制资源的方式显得比较粗旷，也无法限制磁盘 I/O，所以从 kernel 2.6.24 开始，引入了一个新的资源限制的方式——Cgroup。

Cgroup 中控制资源的系统成为 controller，Cgroup 提供了如下的 controller 来控制 CPU、内存、块设备、进程资源等。

- CPU/cpuacct/cpuset
- memory
- blkio
- device
- freezer
- net_cls

当 Cgroup 进程启动时，它会在系统中产生一个挂载点，在这个挂载点里含有 Cgroup 一切的配置，这些配置可以通过 echo <value> 的方式直接修改，也可以通过编译配置文件 /etc/cgconfig.conf 和 /etc/cgrules.conf 来让配置持久化。

Cgroup 的配置选项也颇多，下面使用实战的方式来理解 Cgroup 的原理，并且了解如何使用 Cgroup。

1. 安装 Cgroup

安装 libcgroup 包，启动 cgconfig 服务。会发现根目录中出现了 /cgroup 这个目录，同时里面还有一些子目录，如下：

```
[root@systemtap ~]# yum -y install libcgroup
[root@systemtap ~]# /etc/init.d/cgconfig start
[root@systemtap ~]# ls /cgroup/
blkio  cpu  cpuacct  cpuset  devices  freezer  memory  net_cls
```

2. 限制 Apache 内存使用

Cgroup 有两个主要的配置文件：/etc/cgconfig.conf 和 /etc/cgrules.conf，cgconfig.conf 用来定义资源限制的规则，比如可以使用多少内存、磁盘 I/O 等，cgrules.conf 用来定义哪些程序，或者哪些用户使用哪一种限制规则的。这里以限制 Apache 内存为例讲解基础配置。

在 /etc/cgconfig.conf 中，默认有一个 mount 规则，mount 规则的意义在于默认情况下需要对哪些资源做限制，比如不需要对磁盘 I/O 做限制，就可以去掉 blkio 这行。示例如下：

```
mount {
    cpuset  = /cgroup/cpuset;
    cpu     = /cgroup/cpu;
    cpuacct = /cgroup/cpuacct;
    memory  = /cgroup/memory;
```

```
    devices = /cgroup/devices;
    freezer = /cgroup/freezer;
    net_cls = /cgroup/net_cls;
    blkio   = /cgroup/blkio;
}
```

规则的语法是 group <name> { <controller> {<param name> = <param value>;} }，这里要限制 Apache 可以使用的内存为 1MB，因此可在 /etc/cgconfig.conf 里写上如下代码：

```
group httpd {
    memory {
        memory.limit_in_bytes=102400;
        memory.swappiness=0;
    }
}
```

在 cgrules.conf 中加入如下一行，语义为所有用户执行 apachectl 这个命令时应用 cgroup memory 这个 controller，规则是 cgconfig.conf 中的 httpd。

```
*:/usr/sbin/apachectl    memory  httpd
```

然后，重启服务让配置生效。

```
[root@systemtap ~]# /etc/init.d/cgconfig restart
[root@systemtap ~]# /etc/init.d/cgred restart
```

理论上来说，1MB 内存是无法运行 Apache 服务的，我们看看 Apache 究竟能不能启动。

使用 apachectl 命令启动 Apache 的时候，就发现这个进程被 kill 掉了，因为 Apache 申请的虚拟内存超过了 1MB 的大小，触发了系统的 OOM！

```
[root@systemtap ~]# apachectl restart
Killed
```

在日志中，可以很明确地看到 apachectl 被 kill 掉的全部行为。

```
Jun 14 15:56:12 systemtap kernel: apachectl invoked oom-killer: gfp_mask=0xd0,
    order=0, oom_adj=0, oom_score_adj=0
Jun 14 15:56:12 systemtap kernel: [<ffffffff81127782>] ? oom_kill_process+0x82/
    0x2a0
Jun 14 15:56:12 systemtap kernel: Task in /httpd killed as a result of limit of
    /httpd
Jun 14 15:56:12 systemtap kernel: Memory cgroup out of memory: Kill process 2070
    (apachectl) score 1000 or sacrifice child
Jun 14 15:56:12 systemtap kernel: Killed process 2070, UID 0, (apachectl) total-
    vm:106072kB, anon-rss:116kB, file-rss:640kB
```

尝试将内存扩大到 10MB，看看 Apache 能不能启动。

将 memory.limit 的值设置为 10 485 760，然后重启 cgconfig 服务。

```
group httpd {
    memory {
```

```
        memory.limit_in_bytes=10485760;
        memory.swappiness=0;
    }
}
```

此时可以看到 Apache 服务启动了，80 端口在正常的监听中。

```
[root@systemtap ~]# /etc/init.d/cgconfig restart
Stopping cgconfig service:                                 [  OK  ]
Starting cgconfig service:                                 [  OK  ]
[root@systemtap ~]# apachectl start
[root@systemtap ~]# netstat -nltp | grep 80
tcp        0      0 :::80                   :::*        LISTEN      2110/httpd
```

现在使用 ps 命令查看在 cgroup 中 httpd 进程的状态，可以看到 httpd 进程应用到了 memory 这个 controller 上。

```
[root@systemtap ~]# ps -eo pid,cgroup,cmd | grep  httpd
    2110 blkio:/;net_cls:/;freezer:/;devices:/;memory:/httpd;cpuacct:/;cpu:/;cpuset:/
        /usr/sbin/httpd -k start
    2111 blkio:/;net_cls:/;freezer:/;devices:/;memory:/httpd;cpuacct:/;cpu:/;cpuset:/
        /usr/sbin/httpd -k start
    2112 blkio:/;net_cls:/;freezer:/;devices:/;memory:/httpd;cpuacct:/;cpu:/;cpuset:/
        /usr/sbin/httpd -k start
    2113 blkio:/;net_cls:/;freezer:/;devices:/;memory:/httpd;cpuacct:/;cpu:/;cpuset:/
        /usr/sbin/httpd -k start
    2114 blkio:/;net_cls:/;freezer:/;devices:/;memory:/httpd;cpuacct:/;cpu:/;cpuset:/
        /usr/sbin/httpd -k start
    2115 blkio:/;net_cls:/;freezer:/;devices:/;memory:/httpd;cpuacct:/;cpu:/;cpuset:/
        /usr/sbin/httpd -k start
    2116 blkio:/;net_cls:/;freezer:/;devices:/;memory:/httpd;cpuacct:/;cpu:/;cpuset:/
        /usr/sbin/httpd -k start
    2117 blkio:/;net_cls:/;freezer:/;devices:/;memory:/httpd;cpuacct:/;cpu:/;cpuset:/
        /usr/sbin/httpd -k start
    2118 blkio:/;net_cls:/;freezer:/;devices:/;memory:/httpd;cpuacct:/;cpu:/;cpuset:/
        /usr/sbin/httpd -k start
    2122  blkio:/;net_cls:/;freezer:/;devices:/;memory:/;cpuacct:/;cpu:/;cpuset:/
        grep httpd
```

3. 限制磁盘 I/O

在现实环境中，可能需要对一些写入优先级不高的进程做 I/O 限制，比如一些级别很低的文件备份时，这里用 dd 命令来演示如何限制磁盘读写。

首先在 /etc/cgconfig.conf 中添加如下规则：

```
group diskio {
    blkio {
        blkio.throttle.read_bps_device="8:0 1048576";
        blkio.throttle.write_bps_device="8:0 20480";
    }
}
```

在 diskio 这个规则中，使用 blkio.throttle.read/write_bps_device 来控制磁盘 I/O 速率，这里还可以使用 blkio.throttle.read_iops_device 来控制 IOPS，以达到同样的目的。

blkio.throttle.read/write_bps_device 后面的参数分别是磁盘号和限制的值。8：0 是 /dev/sda 这个块设备号，用 ls -l 命令查看，可知 1 048 576 是最大读取速率，这里则是 1MB。

```
[root@systemtap ~]# ll /dev/sda
brw-rw---- 1 root disk 8, 0 Jun 14 21:04 /dev/sda
```

然后在 /etc/cgrules.conf 中添加一个应用规则：

```
*:/bin/dd       blkio   diskio
```

一切配置完成，重启 cgconfig 和 cgred 两个服务，准备开始测试。

首先，测试磁盘读 I/O 的限制。打开两个终端，一个终端中开启 iotop，另外一个终端中运行如下命令：

```
[root@systemtap ~]# dd if=/dev/sda of=/dev/null
```

此时可在 iotop 上看到，DISK READ 被精确地限制在了 1000KB 左右，测试成功。

```
Total DISK READ: 1001.87 K/s | Total DISK WRITE: 0.00 B/s
    TID  PRIO  USER     DISK READ  DISK WRITE  SWAPIN     IO>    COMMAND
 1664 be/4 root     1001.87 K/s    0.00 B/s  0.00 % 97.97 % dd if=/de~=/dev/null
    1 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % init
```

终止 dd 命令之后，也可以看到 dd 命令显示出每秒只能读取 1MB。

```
[root@systemtap ~]# dd if=/dev/sda of=/dev/null
^C252473+0 records in
252472+0 records out
129265664 bytes (129 MB) copied, 123.968 s, 1.0 MB/s
```

成功测试了磁盘读 I/O 之后，再来测试磁盘写 I/O。因为 CentOS 6 内核中的 blkio 只支持磁盘 I/O 的 sync 和 direct 两种方式，不支持 buffer 方式。所以这里将使用 dd 的 direct I/O 方式来测试。因为 direct I/O 速率较慢，所以这里将写速率限制到 20KB，以便能看到明显的效果。

运行如下命令：

```
[root@systemtap ~]# dd if=/dev/zero of=/tmp/file oflag=direct
```

从 iotop 上，几乎可以看到 DISK WRITE 被控制在了 20KB，说明采用的方式成功了。

```
Total DISK READ: 0.00 B/s | Total DISK WRITE: 21.67 K/s
    TID  PRIO  USER     DISK READ  DISK WRITE  SWAPIN     IO>    COMMAND
    1702 be/4 root        0.00 B/s   21.67 K/s  0.00 % 99.99 % dd if=/de~
```

终止 dd 命令之后，发现平均写约为 10KB，低于设置的 20K。主要原因在于磁盘写 I/O 机制比读 I/O 要复杂很多，在 2.6.32 这个内核中 blkio 还不能非常精确地控制好写 I/O。

```
[root@systemtap ~]# dd if=/dev/zero of=/tmp/file oflag=direct
^C577+0 records in
577+0 records out
295424 bytes (295 kB) copied, 27.5842 s, 10.7 kB/s
```

4. 限制虚拟机 CPU

虚拟机的 VCPU 一般来说是由 libvirtd 自动分配的，但是有时候为了减少虚拟化环境中的 CPU 切换，会将 VCPU 绑定在某一个物理 CPU 的核上。

首先，查看虚拟机 VCPU 现在的信息，如下：

```
[root@kvm ~]# virsh vcpuinfo guest
VCPU:           0
CPU:            5
State:          running
CPU time:       19927.5s
CPU Affinity:   yyyyyyyyyyyyyyyy

VCPU:           1
CPU:            8
State:          running
CPU time:       17281.8s
CPU Affinity:   yyyyyyyyyyyyyyyy
```

可以看到，两个 VCPU 分别在物理 CPU 的第 5 个和第 8 个核心上。

此时，重启 cgconfig 服务，然后重启 libvirtd 服务及虚拟机，保证 cgconfig 载入了虚拟机配置。

```
[root@kvm ~]# /etc/init.d/cgconfig restart
Stopping cgconfig service:                                 [  OK  ]
Starting cgconfig service:                                 [  OK  ]
[root@kvm ~]# /etc/init.d/libvirtd restart
Stopping libvirtd daemon:                                  [  OK  ]
Starting libvirtd daemon:                                  [  OK  ]

[root@kvm ~]# virsh destroy guest
[root@kvm ~]# virsh start guest
```

此时会看到 Cgroup 目录中产生了虚拟机的相应配置文件：

```
[root@kvm ~]# ls -l /cgroup/cpuset/libvirt/qemu/guest/
total 0
--w--w--w- 1 root root 0 Jun 15 15:45 cgroup.event_control
-r--r--r-- 1 root root 0 Jun 15 15:45 cgroup.procs
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.cpu_exclusive
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.cpus
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.mem_exclusive
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.mem_hardwall
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.memory_migrate
-r--r--r-- 1 root root 0 Jun 15 15:45 cpuset.memory_pressure
```

```
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.memory_spread_page
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.memory_spread_slab
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.mems
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.sched_load_balance
-rw-r--r-- 1 root root 0 Jun 15 15:45 cpuset.sched_relax_domain_level
drwxr-xr-x 2 root root 0 Jun 15 15:45 emulator
-rw-r--r-- 1 root root 0 Jun 15 15:45 notify_on_release
-rw-r--r-- 1 root root 0 Jun 15 15:45 tasks
drwxr-xr-x 2 root root 0 Jun 15 15:45 vcpu0
drwxr-xr-x 2 root root 0 Jun 15 15:45 vcpu1
```

这里的重点是 vcpu0 和 vcpu1 两个文件，可以看到里面的内容是 0~15，意思就是这两个 VCPU 可以绑定在物理 CPU 的 0～15 个核的任意一个上。

```
[root@kvm ~]# cat /cgroup/cpuset/libvirt/qemu/guest/vcpu0/cpuset.cpus
0-15
[root@kvm ~]# cat /cgroup/cpuset/libvirt/qemu/guest/vcpu1/cpuset.cpus
0-15
```

下面使用 echo 的方式向文件中直接写入想绑定的物理 CPU 的核心上。

```
[root@kvm ~]# echo 9 > /cgroup/cpuset/libvirt/qemu/guest/vcpu0/cpuset.cpus
[root@kvm ~]# cat /cgroup/cpuset/libvirt/qemu/guest/vcpu0/cpuset.cpus 9
[root@kvm ~]# echo 10 > /cgroup/cpuset/libvirt/qemu/guest/vcpu1/cpuset.cpu
[root@kvm ~]# cat /cgroup/cpuset/libvirt/qemu/guest/vcpu1/cpuset.cpus 10
```

此时就可以看到 VCPU 已经被绑定到相应的 CPU 上了，如下：

```
[root@kvm ~]# virsh vcpuinfo guest
VCPU:           0
CPU:            9
State:          running
CPU time:       35.5s
CPU Affinity:   ---------y------

VCPU:           1
CPU:            10
State:          running
CPU time:       12.6s
CPU Affinity:   ----------y-----
```

这是使用 echo 方式直接使虚拟机生效的方式，如果希望系统重启之后依然生效，那必须要写入 cgconfig.conf 配置文件中，控制 CPU 的 controller 是 cpuset，整个配置的写法如下：

```
group libvirt {
    cpuset {
        cpuset.cpus=0-15;
        cpuset.mems=0;
    }
}
```

```
group libvirt/qemu {
    cpuset {
        cpuset.cpus=0-15;
           cpuset.mems=0;
    }
}

group libvirt/qemu/guest {
    cpuset {
        cpuset.cpus=8-13;
    }
}
group libvirt/qemu/guest/vcpu0 {
    cpuset {
        cpuset.cpus=9;
    }
}

group libvirt/qemu/guest/vcpu1 {
    cpuset {
        cpuset.cpus=10;
    }
}
```