Linux CPU 占用率原理与精确度分析

http://ilinuxkernel.com

1 CPU占用率计算原理

1.1 相关概念

在 Linux/Unix 下，CPU 利用率分为用户态、系统态和空闲态， 分别表示 CPU 处于用户态执行的时间，系统内核执行的时间，和空闲系统进程执行的时间。

下面是几个与 CPU 占用率相关的概念。

- **CPU 利用率**

  CPU 的使用情况。

- **用户时间(User time)**

  表示 CPU 执行用户进程的时间，包括 nices 时间。通常期望用户空间 CPU 越高越好。

- **系统时间(System time)**

  表示 CPU 在内核运行时间，包括 IRQ 和 softirq 时间。系统 CPU 占用率高，表明系统某部分存在瓶颈。通常值越低越好。

- **等待时间(Waiting time)**

  CPI 在等待 I/O 操作完成所花费的时间。系统部应该花费大量时间来等待 I/O 操作，否则就说明 I/O 存在瓶颈。

- **空闲时间(Idle time)**

  系统处于空闲期，等待进程运行。

- **Nice 时间(Nice time)**

  系统调整进程优先级所花费的时间。

- **硬中断处理时间(Hard Irq time)**

  系统处理硬中断所花费的时间。

- **软中断处理时间(SoftIrq time)**

系统处理软中断中断所花费的时间。

- **丢失时间(Steal time)**

被强制等待（involuntary wait）虚拟 CPU 的时间，此时 hypervisor 在为另一个虚拟处理器服务。

下面是我们在 top 命令看到的 CPU 占用率信息及各项值含义。

Cpu(s): 0.2%us, 0.2%sy, 0.0%ni, 99.2%id, 0.5%wa, 0.0%hi, 0.0%si, 0.0%st

us: User time

sy: System time

ni: Nice time

id: Idle time

wa: Waiting time

hi: Hard Irq time

si: SoftIrq time

st: Steal time

1.2 CPU占用率计算

Linux CPU 占用率计算，都是根据/proc/stat 文件内容计算而来，下面是 stat 文件内容样例，内核版本不同，会稍有不同，但内容基本一致。

```
[root@linux_driver ~]# cat /proc/stat
cpu  661733 468 503925 233055573 548835 14244 15849 0
cpu0 64228 64 125723 57989179 504273 12397 4420 0
cpu1 308241 0 119618 58264165 7902 66 60 0
cpu2 167598 226 151336 58340033 28829 1663 10563 0
cpu3 121664 176 107246 58462194 7829 117 804 0
intr 611601623 587093629 3 0 0 2 0 3 0 1 0 0 0 4 0 5257161 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 11150346 0 0 0 0 0 0 0 293531 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 7806901 0 0 0 0 0 0 0 40 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0
ctxt 1887995917
btime 1244449131
processes 1938183
procs_running 1
procs_blocked 0
```

CPU 信息，cpu 为总的信息，cpu0 ... cpun 为各个具体 CPU 信息

cpu 661733 468 503925 233055573 548835 14244 15849 0

上面共有 8 个值（**单位：ticks**），分别为：

User time， 661733　　Nice time， 468

System time， 503925　　Idle time，233055573

Waiting time，548835　　Hard Irq time， 14244

SoftIRQ time，15849　　Steal time，0

CPU 占用率计算公式如下：

CPU 时间=user+system+nice+idle+iowait+irq+softirq+Stl

%us=(User time + Nice time)/CPU 时间*100%

%sy=(System time + Hard Irq time +SoftIRQ time)/CPU 时间*100%

%id=(Idle time)/CPU 时间*100%

%ni=(Nice time)/CPU 时间*100%

%wa=(Waiting time)/CPU 时间*100%

%hi=(Hard Irq time)/CPU 时间*100%

%si=(SoftIRQ time)/CPU 时间*100%

%st=(Steal time)/CPU 时间*100%

2 CPU占用率内核实现

下面以 RHEL6 内核源码版本 2.6.32-220.el6 x86_64 为例，来介绍内核源码实现。/proc/stat 文件的创建由函数 proc_stat_init（）实现，在文件 fs/proc/stat.c 中，在内核初始化时调用。/proc/stat 文件相关函数时间均在 stat.c 文件中。

对/proc/stat 文件的读写方法为 proc_stat_operations。

```
00152: static const struct file_operations proc_stat_operations = {
00153:         .open           = stat_open,
00154:         .read           = seq_read,
00155:         .llseek         = seq_lseek,
00156:         .release    = single_release,
00157: };
00158:
```

打开文件函数 stat_open（），函数首先申请大小为 size 的内存，来存放临时数据（也是我们看到的 stat 里的最终数据）。

```
00128: static int stat_open(struct inode *inode, struct file *file)
00129: {
00130:         unsigned size = 4096 * (1 + num_possible_cpus() / 32);
00131:         char *buf;
00132:         struct seq_file *m;
00133:         int res;
00134:
00135:         /* don't ask for more than the kmalloc() max size, currently 128 KB */
```

```
00136:      if (size > 128 * 1024)
00137:           size = 128 * 1024;
00138:      buf = kmalloc(size, GFP_KERNEL);
00139:      if (! buf)
00140:           return - ENOMEM;
00141:
00142:      res = single_open(file, show_stat, NULL);
00143:      if (! res) {
00144:           m = file- >private_data;
00145:           m- >buf = buf;
00146:           m- >size = size;
00147:      } else
00148:           kfree(buf);
00149:      return res;
00150: } ?   end stat_open ?
00151:
```

/proc/stat 文件的数据由 show_stat()函数填充。注意 42 行 for_each_possible_cpu(i)循环，是累加计算所有 CPU 的数据，如我们前面的示例看到的/proc/stat 文件中第一行 cpu 值。

cpu *661733 468 503925 233055573 548835 14244 15849 0*

```
00025: static int show_stat(struct seq_file *p, void *v)
00026: {
00027:      int i, j;
00028:      unsigned long jif;
00029:      cputime64_t user, nice, system, idle, iowait, irq, softirq,
steal;
00030:      cputime64_t guest;
00031:      u64 sum = 0;
00032:      u64 sum_softirq = 0;
00033:      unsigned int per_softirq_sums[NR_SOFTIRQS] = {0};
00034:      struct timespec boottime;
00035:
00036:      user = nice = system = idle = iowait =
00037:           irq = softirq = steal = cputime64_zero;
00038:      guest = cputime64_zero;
00039:      getboottime(&boottime);
00040:      jif = boottime.tv_sec;
00041:
00042:      for_each_possible_cpu(i) {
00043:           user = cputime64_add(user, kstat_cpu(i).cpustat.user);
00044:           nice = cputime64_add(nice, kstat_cpu(i).cpustat.nice);
00045:           system = cputime64_add(system,
                              kstat_cpu(i).cpustat.system);
00046:           idle = cputime64_add(idle, kstat_cpu(i).cpustat.idle);
```

```
00047:          idle = cputime64_add(idle, arch_idle_time(i));
00048:          iowait = cputime64_add(iowait,
                                kstat_cpu(i).cpustat.iowait);
00049:          irq = cputime64_add(irq, kstat_cpu(i).cpustat.irq);
00050:          softirq = cputime64_add(softirq,
                                kstat_cpu(i).cpustat.softirq);
00051:          steal = cputime64_add(steal, kstat_cpu(i).cpustat.steal);
00052:          guest = cputime64_add(guest,
                                kstat_cpu(i).cpustat.guest);
00053:          sum += kstat_cpu_irqs_sum(i);
00054:          sum += arch_irq_stat_cpu(i);
00055:
00056:          for (j = 0; j < NR_SOFTIRQS; j++) {
00057:                  unsigned int softirq_stat = kstat_softirqs_cpu(j, i);
00058:
00059:                  per_softirq_sums[j] += softirq_stat;
00060:                  sum_softirq += softirq_stat;
00061:          }
00062:  }
00063:  sum += arch_irq_stat();
00064:
00065:  seq_printf(p,
                "cpu  %llu %llu %llu %llu %llu %llu %llu %llu %llu\n",
00066:          (unsigned long long)cputime64_to_clock_t(user),
00067:          (unsigned long long)cputime64_to_clock_t(nice),
00068:          (unsigned long long)cputime64_to_clock_t(system),
00069:          (unsigned long long)cputime64_to_clock_t(idle),
00070:          (unsigned long long)cputime64_to_clock_t(iowait),
00071:          (unsigned long long)cputime64_to_clock_t(irq),
00072:          (unsigned long long)cputime64_to_clock_t(softirq),
00073:          (unsigned long long)cputime64_to_clock_t(steal),
00074:          (unsigned long long)cputime64_to_clock_t(guest));
```

计算总的CPU各个值user、nice、system、idle、iowait、irq、softirq、steal后，就分别计算各个CPU的使用情况（75~100行）。

```
00075:  for_each_online_cpu(i) {
00076:
00077:          /* Copy values here to work around gcc-2.95.3, gcc-2.96 */
00078:          user = kstat_cpu(i).cpustat.user;
00079:          nice = kstat_cpu(i).cpustat.nice;
00080:          system = kstat_cpu(i).cpustat.system;
00081:          idle = kstat_cpu(i).cpustat.idle;
00082:          idle = cputime64_add(idle, arch_idle_time(i));
00083:          iowait = kstat_cpu(i).cpustat.iowait;
00084:          irq = kstat_cpu(i).cpustat.irq;
00085:          softirq = kstat_cpu(i).cpustat.softirq;
00086:          steal = kstat_cpu(i).cpustat.steal;
```

```
00087:          guest = kstat_cpu(i).cpustat.guest;
00088:          seq_printf(p,
00089:              "cpu%d %llu %llu %llu %llu %llu %llu %llu %llu\n",
00090:               i,
00091:               (unsigned long long)cputime64_to_clock_t(user),
00092:               (unsigned long long)cputime64_to_clock_t(nice),
00093:               (unsigned long long)cputime64_to_clock_t(system),
00094:               (unsigned long long)cputime64_to_clock_t(idle),
00095:               (unsigned long long)cputime64_to_clock_t(iowait),
00096:               (unsigned long long)cputime64_to_clock_t(irq),
00097:               (unsigned long long)cputime64_to_clock_t(softirq),
00098:               (unsigned long long)cputime64_to_clock_t(steal),
00099:               (unsigned long long)cputime64_to_clock_t(guest));
00100:      }
```

104 行计算所有 CPU 上中断次数，104~105 行计算 CPU 上每个中断向量的中断次数。注意：/proc/stat 文件中，将所有可能的 NR_IRQS 个中 断向量计数都记录下来，但我们的机器上通过只是用少量的中断向量，这就是看到/proc/stat 文件中，intr 一行后面很多值为 0 的原因。

show_stat（）函数最后获取进程切换次数 nctxt、内核启动的时间 btime、所有创建的进程 processes、正在运行进程的数量 procs_running、阻塞的进程数量 procs_blocked 和所有 io 等待的进程数量。

```
00101:      seq_printf(p, "intr %llu", (unsigned long long)sum);
00102:
00103:      /* sum again ? it could be updated? */
00104:      for_each_irq_nr(j)
00105:          seq_printf(p, " %u", kstat_irqs(j));
00106:
00107:      seq_printf(p,
00108:          "\nctxt %llu\n"
00109:          "btime %lu\n"
00110:          "processes %lu\n"
00111:          "procs_running %lu\n"
00112:          "procs_blocked %lu\n",
00113:          nr_context_switches(),
00114:          (unsigned long)jif,
00115:          total_forks,
00116:          nr_running(),
00117:          nr_iowait());
00118:
00119:      seq_printf(p, "softirq %llu", (unsigned long long)sum_softirq);
00120:
00121:      for (i = 0; i < NR_SOFTIRQS; i++)
00122:          seq_printf(p, " %u", per_softirq_sums[i]);
```

```
00123:      seq_printf(p, "\n");
00124:
00125:      return 0;
00126: } ?   end show_stat ?
00127:
```

3 Linux CPU占用率精确性分析

在使用类似 top 命令，观察系统及各进程 CPU 占用率时，可以指定刷新时间间隔，以及时刷新和实时观察 CPU 占用率。

top 命令默认情况下，是每 3 秒刷新一次。也可以通过 top -d <刷新时间间隔> 来指定刷新频率，如 top -d 0.1 或 top -d 0.01 等。top 执行时，也可以按“s”键，修改时间间隔。

我们可以将 CPU 占用率刷新间隔设置很低，如 0.01 秒。但过低的刷新频率是否能够更准确观察到 CPU 占用率？Linux 系统提供的 CPU 占用率信息是否足够精确？

根据前面分析，我们已知 Linux 是根据/proc/stat 文件的内容来计算 CPU 占用率，也就是精确度和/proc/stat 提供的数据精确度有关。那么

（1）/proc/stat 文件中的内容单位是什么？

（2）多久会刷新/proc/stat 中的数据？

cpu 926 0 4160 5894903 2028 0 7 0 0

cpu0 80 0 473 367723 658 0 3 0 0

3.1 /proc/stat中的数据单位精度

/proc/stat 中 CPU 数据信息，单位是 **ticks**。内核中有个全局变量 jiffies，来记录系统启动以来，经历的 ticks 数量。

cpu1 13 0 200 368639 63 0 0 0 0

ticks（滴答）就是系统时钟中断的时间间隔，该值与内核中 HZ 值有关，即 ticks = 1/HZ。HZ 值的大小，在内核编译时可配置的。某台机器上是 RHEL6.1 内核，配置的 HZ 值为 1000。

[root@ssd boot]# uname -a

Linux ssd 2.6.32-131.0.15.el6.x86_64 #1 SMP Tue May 10 15:42:40 EDT 2011 x86_64 x86_64 x86_64 GNU/Linux

[root@ssd boot]# cat config-2.6.32-131.0.15.el6.x86_64 |grep CONFIG_HZ

 # 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

[root@ssd boot]#

HZ 的值，就是每秒的时钟中断数量。可以观察/proc/interrupts 中时钟中断值变化，来计算 HZ 的值。当 HZ 的值为 1000 时，ticks 的单位即为 1/1000 秒，即 1ms。

Every 5.0s: cat /proc/interrupts |grep LOC
Tue May 15 15:54:22 2012

LOC: 1621246 308599 28013 16995 37126 95699
1159285 2399641 552961 63923 58053

20580 17037 49626 1004223 48133 Local timer interrupts

3.2 CPU利用率统计信息更新

在时钟中断程序中，更新 CPU 利用信息，即每个 ticks 更新一次。include/linux/kernel_stat.h 中，有相应函数接口，专门用来更新 CPU 利用率信息。如 account_user_time（）是更新用户态 CPU 信息。

```
00111: /*
00112:  * Lock/unlock the current runqueue - to extract task statistics:
00113:  */
00114: extern unsigned long long task_delta_exec(struct task_struct *);
00115:
00116: extern void account_user_time(struct task_struct *, cputime_t, cputime_t);
00117: extern void account_system_time(struct task_struct *, int, cputime_t,
```

```
00117:  cputime_t);
00118: extern void account_steal_time(cputime_t);
00119: extern void account_idle_time(cputime_t);
00120:
00121: extern void account_process_tick(struct task_struct *,
int user);
00122: extern void account_steal_ticks(unsigned long ticks);
00123: extern void account_idle_ticks(unsigned long ticks);
```

在内核中有一个 per CPU 变量 kernel_stat，专门用来记录 CPU 利用信息。其定义在 include/linux/kernel_stat.h 中。

```
00039:  DECLARE_PER_CPU(struct kernel_stat, kstat);
00040:
00041: #define kstat_cpu(cpu)      per_cpu(kstat, cpu)
```

每次时钟中断时（ticks），就会更新 kernel_stat 变量中各个成员变量的值。/proc/stat 文件中的值，都是在程序读取时更新，内核并不会主动更新/proc/stat 中的数据。/proc/stat 中的 CPU 信息是通过 kernel_stat 各个成员变量的值计算而来。

3.3 CPU利用率精确性分析

通过前面分析，我们可以得出以下结论：

（1）Linux CPU 占用率是根据/proc/stat 文件中的数据计算而来；

（2）/proc/stat 中的数据精度为 ticks，即 1/HZ 秒；

（3）内核每个 ticks 会更新一次 CPU 使用信息；

（4）CPU 占用率的精度为 1/HZ 秒。

4 Linux CPU占用率是否准确？

有时偶尔会遇到类似问题：在稳定计算压力下，进程 CPU 占用率不稳定；或者特性进程 CPU 占用率明显不准。即在系统切换次数很高时，Linux 的 CPU 利用率计算机制可能不准确。

那么 Linux 的 CPU 利用率计算到底是否准确？若可能不准确，则什么情况下出现这种情况？

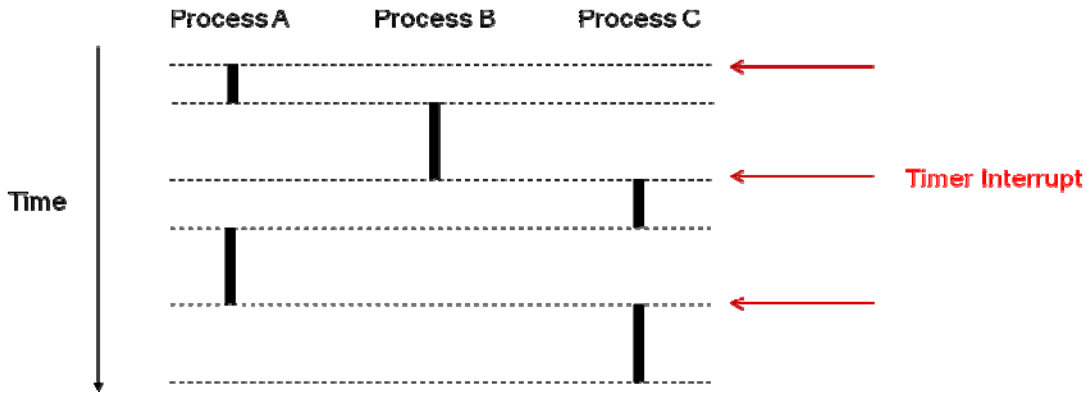
4.1 Linux CPU占用率不准确情形

在前面分析中，Linux 内核是在每次时钟中断时更新 CPU 使用情况，即 1/HZ 秒更新一次。时钟中断时，只会看到当前正在运行的进程信息。以下图为例，红色箭头表示时钟中断（Timer Interrupt）。

第一次中断时，看到进程 A 在运行。但进程 A 运行时间短，进程 B 运行。第二次中断时，进程 C 运行；在第三次中断到来时，再次调度进程 A 执行。第三次此中断时，进程 C 运行。

按照 Linux 内核 CPU 占用率统计方法，在第 1 次和第 2 次中断期间，内核并没有看到进程 B 在运行；于是就漏掉了进程 B 使用 CPU 的信息。同样道理，在第 2 次和第 3 次中断期间，漏掉了进程 B 使用 CPU 的情况。这样，就导致了 Linux 内核 CPU 占用率统计不准确。

发生 CPU 占用率不准确的原因是：**在一个时钟中断周期内，发生了多次进程调度**。时钟中断的精度是 1/HZ 秒。



4.2 top命令CPU使用率准确吗？

只有在一个时钟中断周期内发生多次进程调度，才会出现 CPU 占用率不准的情况。那么 top 命令中 CPU 使用率是否准确与进程调度频率有关。

若 HZ 的值为 250，则 ticks 值为 4ms；若 HZ 值为 1000，则 ticks 值为 1ms。在 HZ 为 250 时，只要进程的调度间隔大于 4ms，CPU 占用率就准确。HZ 为 1000 时，调度间隔大于 1ms，CPU 占用率计算就准确。

进程调度次数少，CPU 占用率就准确；调度时间间隔小于时钟中断，就可能不准确。那么进程调度的时机是怎样的？如何观察进程调度次数？

4.2.1 进程调度时机

- 进程状态转换的时刻：进程终止、进程睡眠

进程要调用 sleep（）或 exit（）等函数进行状态转换，这些函数会主动调用调度程序进行进程调度；

- 当前进程的时间片用完时（current->counter=0）

由于进程的时间片是由时钟中断来更新的

- 设备驱动程序

当设备驱动程序执行长而重复的任务时，直接调用调度程序。在每次反复循环中，驱动程序都检查 need_resched 的值，如果必要，则调用调度程序 schedule()主动放弃 CPU。

- 进程从中断、异常及系统调用返回到用户态时

不管是从中断、异常还是系统调用返回，最终都调用 ret_from_sys_call（），由这个函数进行调度标志的检测，如果必要，则调用调度程序。那么，为什么从系统调用返回时要调用调度程序呢？这当然是从效率考虑。从系统调用返回意味着要离开内核态而返回到用户态，而状态的转换要花费一定的时间，因此，在返回到用户态前，系统把在内核态该处理的事全部做完。

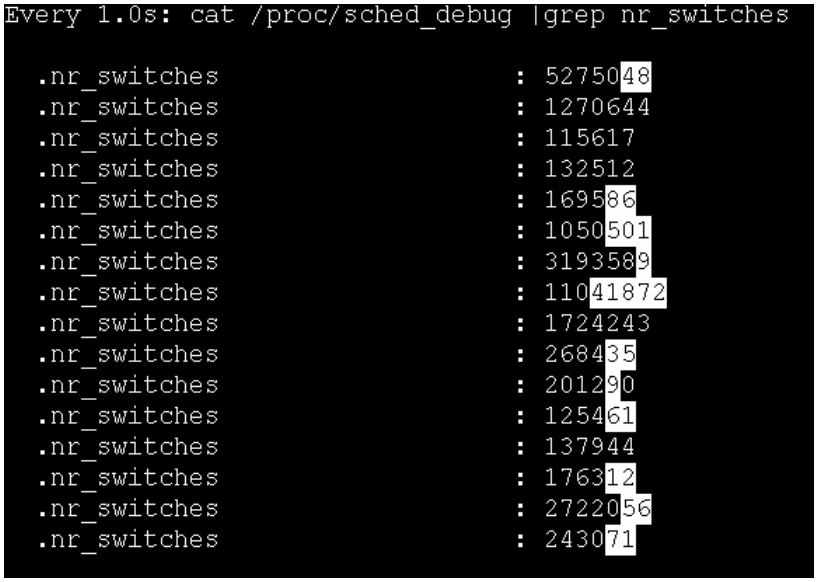
4.2.2 进程调度次数观察

可以通过 vmstat 命令，来观察系统中进程切换次数，cs 域的值就是切换次数。HZ 的值，可以通过内核配置文件来确定，若/proc/config.gz 存在，导出这个文件查看即可。

```
[root@ssd proc]# vmstat 1
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 2  0      0 23818652  14500 173604    0    0     1     0   66  139  0  0 100  0  0
 0  0      0 23818652  14500 173604    0    0     0     0 1043 2139  0  0 100  0  0
 0  0      0 23818520  14500 173604    0    0     0     0 1036 2180  0  0 100  0  0
 0  0      0 23818520  14500 173604    0    0     0     0 1025 2130  0  0 100  0  0
 0  0      0 23819016  14500 173604    0    0     0     0 1046 2201  0  0 100  0  0
 0  0      0 23819016  14500 173604    0    0     0     0 1026 2141  0  0 100  0  0
```

也可以通过查看/proc/sched_debug 文件内容，来观察切换次数（nr_switches）。

[root@ssd proc]# watch -d -n 1 'cat /proc/sched_debug |grep nr_switches'



我们系统中的进程调度真的那么频繁吗？大多数情况下，Linux 中的 CPU 占用率计算机制是准确的。