操作系统和硬件优化

讲师：白丁

主页：https://edu.51cto.com/lecturer/9150494.html

目录

CPU (Central Processing Unit) 优化

CPU 架构

日常我们所使用的软件都要经过 CPU 内部的微指令集来完成。这些指令集的设计主要被分为两种，分别是：精简指令集(RISC)与复杂指令集(CISC) 。

精简指令集(Reduced Instruction Set Computer, RISC)

这种 CPU 的设计中，微指令集较为精简，每个指令的运行时间都很短，完成的动作也很单纯，指令的执行效能较佳；但是若要做复杂的事情，要由多个指令来完成。

在中高端服务器中采用 RISC 指令的 CPU 主要有 IBM 公司的 Power PC 、 SUN 公司的 Sparc、HP 公司的 PA-RISC、ARM（常使用的各厂牌手机、导航系统、网络设备(交换器、路由器等)等，几乎都是使用 ARM 架构的 CPU）。

复杂指令集(Complex Instruction Set Computer, CISC)：

与 RISC 不同的，CISC 在微指令集的每个小指令可以执行一些较低阶的硬件操作，指令数目多而且复杂，每条指令的长度并不相同。因为指令执行较为复杂所以每条指令花费的时间较长，但每条个别指令可以处理的工作较为丰富。常见的 CISC 微指令集 CPU 主要有 AMD、Intel、VIA 等的 x86 架构的 CPU。

小常识:

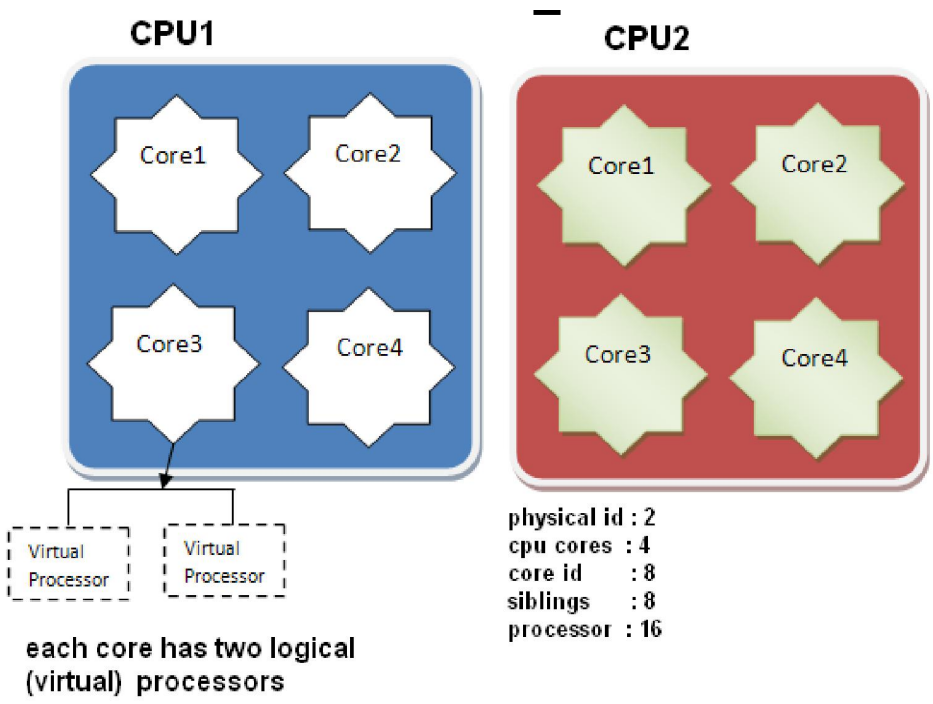
AMD、Intel、VIA 开发出来的 x86 架构 CPU 被大量使用于个人计算机(Personal computer)上面，因此，个人计算机常被称为 x86 架构的计算机！那为何称为 x86 架构呢？这是因为最

早的那颗 Intel 发展出来的 CPU 代号称为 8086，后来依此架构又开发出 80286, 80386……，因此这种架构的 CPU 就被称为 x86 架构了。

在 2003 年以前由 Intel 所开发的 x86 架构 CPU 由 8 位升级到 16、32 位，后来 AMD 依此架构修改新一代的 CPU 为 64 位，为了区别两者的差异，64 位的个人计算机 CPU 又被统称为 x86_64 的架构。64 位 CPU 代表 CPU 一次可以读写 64bits 的数据，32 位 CPU 则是 CPU 一次只能读取 32bits，因此 32 位的 CPU 所能读写的最大数据量大概就是 4GB 左右。

基础概念

- 物理 CPU（路）：主板上实际插入的 CPU 数量
  cat /proc/cpuinfo | grep "physical id" |sort -u | wc -l
- CPU 内核：物理 CPU 的核心数
  cat /proc/cpuinfo | grep "cpu cores" | wc -l
- 逻辑 CPU
  cat /proc/cpuinfo | grep “processor” | wc -l
- 小结：
  CPU 总核数 = 物理 CPU 个数 X 每颗物理 CPU 的核数
  CPU 逻辑数 = 物理 CPU 个数 X 每颗物理 CPU 的核数 X 超线程数



CPU 最大性能模式

操作系统和 CPU 硬件配合，系统不繁忙的时候，为了节约电能和降低温度，它会将 CPU

降频进入节能模式。在 server 环境的 CPU 一定要关闭节能模式，节能模式不适应于服务器环境。关闭 CPU 的节能模式有两种方法：

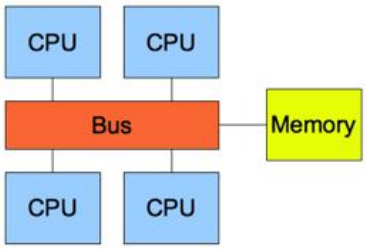
1) 在 BIOS 中进行设置，彻底关闭；
2) 关闭 Linux 中的服务 cpuspeed 和 irqbalance；

> 提示:
>
> cpuspeed 就是负责 CPU 节能的后台服务；而 irqbalance 在 cpuspeed 将某个或某几个 CPU 调节进入休眠模式时，它负责将中断发送到没有休眠的 CPU。关闭 irqbalance 会将所有中断均衡的发送到所有 cpu.

SMP 和 NUMA

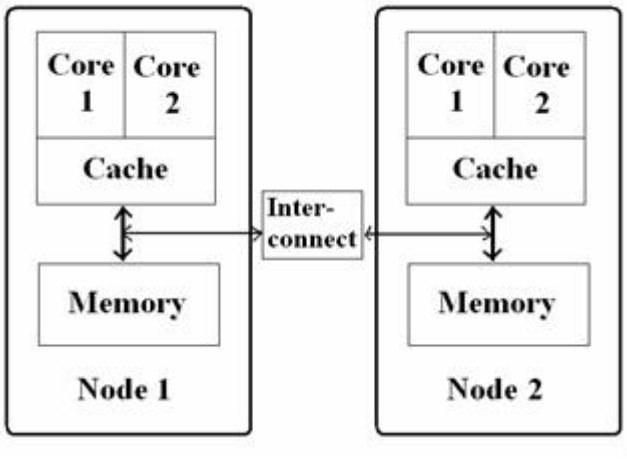
- SMP：称为共享内存访问 CPU(Shared Memory Mulpti Processors)，也称为对称型 CPU 架构（Symmetry Multi Processors)

  SMP 架构的 CPU 内部没有绑定内存，所有的 CPU 争用一个总线来访问所有共享的内存，优点是资源共享，而缺点是总线争用激烈。随着 PC 服务 器上的 CPU 数量变多（不仅仅是 CPU 核数），总线争用的弊端慢慢越来越明显，于是 Intel 在 Nehalem CPU 上推出了 NUMA 架构，而 AMD 也推出了基于相同架构的 Opteron CPU。



- NUMA：非统一内存访问（Non Uniform Memory Access）

  NUMA 引入了 node 的概念，node 内部有多个 CPU，以及绑定的内存。每个 node 的 CPU 和内存一般是相等。NUMA 架构中内存和 CPU 的关系如下图所示：



  NUMA 架构的提出是为了适应多 CPU，可以看到 node 内部的 CPU 对内存的访问分成了两种情况：

  1. 对 node 内部的内存的访问，一般称为 local access，显然访问速度是最

快的；

2. 对其它 node 中的内存的访问，一般称为 remote access，因为要通过 inter-connect，所以访问速度会慢一些；

CPU 和内存是绑定而形成一个 node，那么就涉及到 CPU、内存如何分配的：

- ✓ NUMA 的 CPU 分配策略有 cpunodebind、physcpubind。cpunodebind 规定进程运行在某几个 node 之上，而 physcpubind 可以更加精细地规定运行在哪些核上。
- ✓ NUMA 的内存分配策略有 localalloc、preferred、membind、interleave。
  1. Localalloc 规定进程从当前 node 上请求分配内存；
  2. Preferred 比较宽松地指定了一个推荐的 node 来获取内存，如果被推荐的 node 上没有足够内存，进程可以尝试别的 node。
  3. Membind 可以指定若干个 node，进程只能从这些指定的 node 上请求分配内存。
  4. Interleave 规定进程从所有 node 上以循环(RR)算法交织地请求分配内存，达到随机均匀的从各个 node 中分配内存的目的。

- NUMA 架构导致的问题——SWAP

因为 NUMA 架构的 CPU，默认采用的是 localalloc 的内存分配策略，运行在本 node 内部 CPU 上的进程，会从本 node 内部的内存上分配内存，如果内存不足，则会导致 swap 的产生，严重影响性能！它不会向其它 node 申请内存。这是他最大的问题。

为了避免 SWAP 的产生，一定要将 NUMA 架构 CPU 的内存分配策略设置为：interleave；这样设置的话，任何进程的内存分配，都会随机向各个 node 申请，虽然 remote access 会导致一点点的性能损失，但是他杜绝了 SWAP 导致的严重的性能损失。所以 interleave 其实是将 NUMA 架构的各个 node 中的内存，又重新虚拟成了一个共享的内存，但是和 SMP 不同的是，因为每两个 node 之间有 inter-connect ，所以又避免了 SMP 架构总线争用的缺陷。

- NUMA 的关闭方式

1. BIOS 中关闭 NUMA 设置
2. 在操作系统中关闭 (根据版本关闭方式不同)
3. 在应用级别指定(数据库)

示例：

```
[mysql@db1 ~]$ numactl –hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 16 17 18 19 20 21 22 23
node 0 size: 131037 MB
node 0 free: 3019 MB
node 1 cpus: 8 9 10 11 12 13 14 15 24 25 26 27 28 29 30 31
node 1 size: 131071 MB
node 1 free: 9799 MB
node distances:
node 0 1
```

```
 0: 10 20
 1: 20 10
[mysql@db1 ~]$
```

假设我要执行一个 java param 命令，此命令需要 120G 内存，一个 python param 命令，需要 16G 内存。最好的优化方案时 python 在 node0 中执行，而 java 在 node1 中执行，那命令是：

```
[mysql@db1 ~]$ numactl  --cpunodebind=0  --membind=0  python param
[mysql@db1 ~]$ numactl  --cpunodebind=1  --membind=0,1  java param

[mysql@db1~]$numactl --interleave=all mysqld --defaults-file=/mysql/3306/my.cnf &
```

查看 CPU 状态

vmstat

```
[root@db1 ~]# vmstat 2
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b    swpd   free    buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0   16640  16192       0 235368    0    0     4     3   54   70  0  1 99  0  0
 0  0   16640  16188       0 235368    0    0     0     0   63   84  0  1 99  0  0
 0  0   16640  16188       0 235368    0    0     0    13   67   88  0  1 99  0  0
 1  0   16640  16188       0 235368    0    0     0     0   62   80  0  1 99  0  0
```

输出信息简介：

Procs

- r: 等待运行的进程数。>逻辑 CPU 数量
- b: 处在非中断睡眠状态的进程数

Memory

- swpd: 虚拟内存使用情况，单位：KB
- free: 空闲的内存，单位 KB
- buff: 被用来做为缓存的内存数，单位：KB

Swap

- si: 从磁盘交换到内存的交换页数量，单位：KB/秒
- so: 从内存交换到磁盘的交换页数量，单位：KB/秒

IO

- bi: 发送到块设备的块数，单位：块/秒
- bo: 从块设备接收到的块数，单位：块/秒

System

in: 每秒的中断数，包括时钟中断

cs: 每秒的环境（上下文）切换次数

CPU

按 CPU 的总使用百分比来显示

us: CPU 使用时间

sy: CPU 系统使用时间

id: 闲置时间

知识点补充：进程状态

1) TASK_RUNNING: 正在运行或处于就绪状态
2) TASK_STOPPED: 进程被外部程序暂停
3) TASK_INTERRUPTIBLE: 进程会睡眠直到某个条件变为真，比如说产生一个硬件中断、释放进程正在等待的系统资源或是传递一个信号都可以是唤醒进程的条件
4) TASK_UNINTERRUPTIBLE: 不可中断睡眠状态与可中断睡眠状态类似，但是它有一个例外，那就是把信号传递到这种睡眠状态的进程不能改变它的状态，也就是说它不响应信号的唤醒。
5) TASK_ZOMBIE: 进程资源用户空间被释放，但内核中的进程资源并没有释放，等待父进程回收。

注：通常等待 IO 设备、等待网络的时候，进程会处于 uninterruptible sleep 状态。

pidstat

默认是所有活动进程的 CPU 信息：

```
[root@db1 ~]# pidstat 2
Linux 3.10.0-862.el7.x86_64 (db1)       2019 年 08 月 31 日   _x86_64_        (1 CPU)

10 时 08 分 22 秒   UID       PID    %usr %system  %guest    %CPU   CPU  Command
10 时 08 分 24 秒     0       387    0.00    0.51    0.00    0.51     0  xfsaild/dm-0
10 时 08 分 24 秒  1000     11433    0.00    0.51    0.00    0.51     0  mysqld
10 时 08 分 24 秒     0     15326    0.00    2.56    0.00    2.56     0  kworker/0:0
10 时 08 分 24 秒     0     15353    0.51    1.03    0.00    1.54     0  pidstat
```

查看进程的线程信息：

```
 [root@db1 ~]# pidstat -p 11433 -t 2
Linux 3.10.0-862.el7.x86_64 (db1)       2019 年 08 月 31 日   _x86_64_        (1 CPU)

10 时 22 分 07 秒   UID      TGID       TID    %usr %system  %guest    %CPU   CPU  Command
10 时 22 分 09 秒  1000     11433         -    0.00    0.50    0.00    0.50     0  mysqld
10 时 22 分 09 秒  1000         -     11433    0.00    0.00    0.00    0.00     0  |__mysqld
10 时 22 分 09 秒  1000         -     11434    0.00    0.00    0.00    0.00     0  |__mysqld
10 时 22 分 09 秒  1000         -     11435    0.00    0.00    0.00    0.00     0  |__mysqld
10 时 22 分 09 秒  1000         -     11436    0.00    0.00    0.00    0.00     0  |__mysqld
10 时 22 分 09 秒  1000         -     11437    0.00    0.00    0.00    0.00     0  |__mysqld
10 时 22 分 09 秒  1000         -     11438    0.00    0.00    0.00    0.00     0  |__mysqld
```

```
10 时 22 分 09 秒   1000         -      11439    0.00    0.00    0.00    0.00    0   |__mysqld
10 时 22 分 09 秒   1000         -      11440    0.00    0.00    0.00    0.00    0   |__mysqld
```

其他命令

mpstat: 命令查看多核 CPU 中每个 CPU 核心的状态
sar : sar -q 查看 cpu 队列，任务队列，以及负载
top：

CPU 压力测试

```
下面命令会创建 CPU 负荷，方法是通过压缩随机数据并将结果发送到 /dev/null：
cat /dev/urandom | gzip -9 > /dev/null
如果想要更大的负荷，或者系统有多个核，那么只需要对数据进行压缩和解压就行了，像这样：
cat /dev/urandom | gzip -9 | gzip -d | gzip -9 | gzip -d > /dev/null
```

通过 vmstat、pidstat 定位占用 CPU 最多的进程(线程)。

linux 内存管理

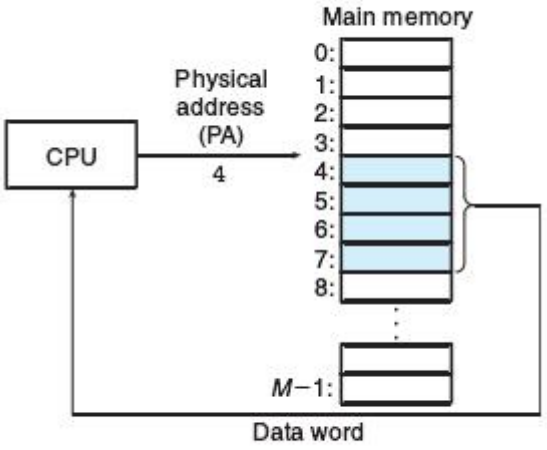
寻址与地址空间

在实际介绍虚拟内存的功能与实现之前我们必须先来大致了解一下**物理寻址，虚拟寻址，物理地址空间，虚拟地址空间**的概念。

物理寻址与虚拟寻址

计算机系统的主存(物理内存)被组织成一个由 M 个连续的字节大小的单元组成的数组，每个字节都有一个唯一的物理地址。第一个字节物理地址为 0,下一个字节为 1，再下一个为 2,以此类推。不带任何存储器抽象的直接使用物理地址的方式就称作**物理寻址(Physical addressing)。**

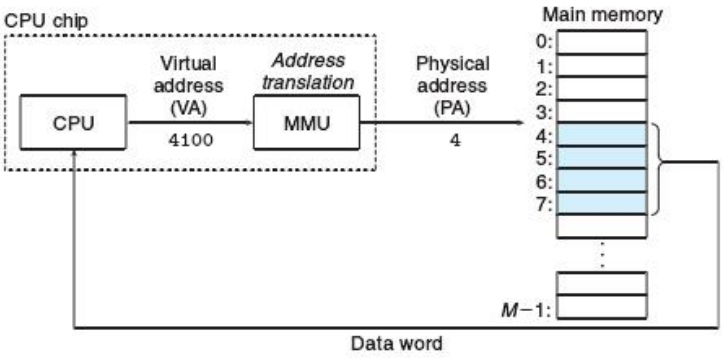
Figure 9.1
A system that uses physical addressing.



如上图展示了一个物理寻址的示例。该示例的上下文是一条加载指令，它读取从物理地

址 4 处开始的 4 字节。当 CPU 执行这条加载指令时，会生成一个有效的物理地址，通过内存总线，把它传递给主存，主存取出物理地址 4 处开始的 4 字节，并将它返回给 CPU。早期的 PC 使用的是物理寻址，现代也有部分具有特殊用途的计算机系统会采用这种寻址方式。但是这种寻址方式对于现代的多道程序设计系统却不适用。所以现代处理器使用的是一种称为**虚拟寻址(Virtual addressing)：**

Figure 9.2
A system that uses virtual addressing.



CPU 通过生成一个虚拟地址(Virtual Address,VA)来访问主存(物理内存)。这个虚拟地址在被送到内存之前先被转换为了适当的物理地址。将一个虚拟地址转换成物理地址的过程叫做地址翻译。该地址翻译由一个 CPU 芯片上的叫做内存管理单元(Memory Management Unit,MMU)的专用硬件完成。MMU 利用存放在主存中的查询表来动态的翻译虚拟地址。这个查询表的内容由操作系统管理，并且整个地址翻译的过程是系统自己完成的，不需要程序员操作。

物理地址空间与虚拟地址空间

地址空间是一个非负整数地址的有序集合.如: {0,1,2,3,...}。

如果地址空间中的整数是连续的，我们说它是一个线性地址空间(在这里我们假设地址空间总是连续的)。在一个带有虚拟内存的系统中，CPU 从一个有 N=2^n 个地址的地址空间中生成虚拟地址，这个地址空间称为**虚拟地址空间(virtual address space):**
$\{0, 1, 2, \ldots, N-1\}$
一个地址空间的大小是由表示最大地址所需要的位数来描述的。例如，一个包含 N=2ˆn 个地址的虚拟地址空间就叫做 n 位地址空间。
一个系统还有**物理地址空间(physical address space)**,对应与系统中物理内存的 M 个字节: {0,1,2,...,M-1}

现在应该要建立这样一个认知：主存(物理内存)中的每字节都有一个选自虚拟地址空间的虚拟地址和一个选自物理地址空间的物理地址。

作为缓存的实现

虚拟内存提供的一个重要能力就是:虚拟内存将主存(物理内存)看作是存储在磁盘上的

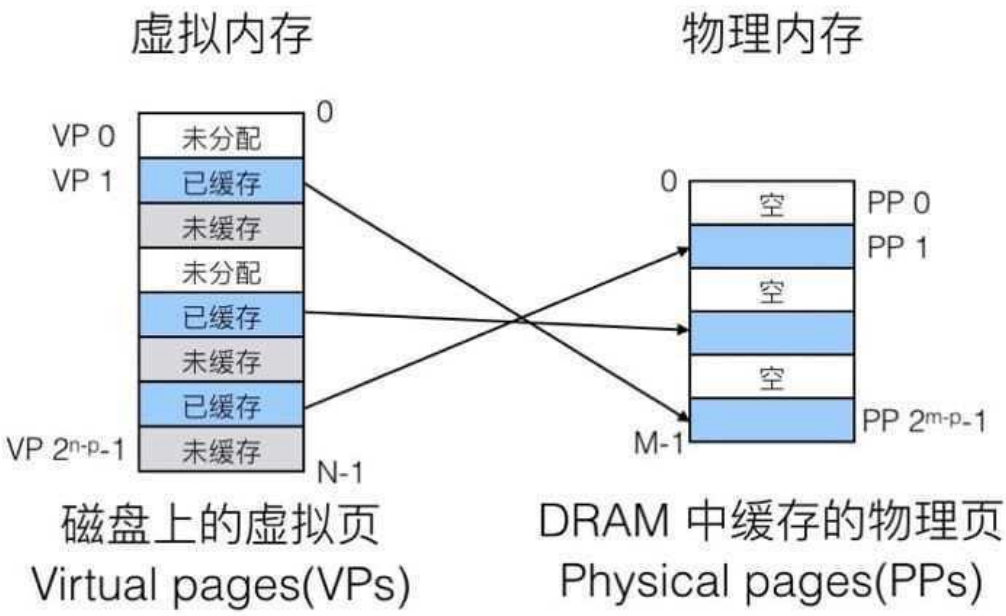
地址空间的缓存。

什么是虚拟内存

在概念上来讲，虚拟内存被组织为由存放在磁盘上的 N 个连续的字节大小的单元组成的数组。每个字节都有唯一的虚拟地址，作为到数组的索引(现在我们知道了虚拟内存虽然叫做内存，但其实它并不在内存中，而是磁盘中的一块空间)。虚拟内存与物理内存构成了一个缓存系统。虚拟内存位于较低层，物理内存位于较高层。虚拟内存上的数据可以被缓存到主存中。VM 系统将虚拟内存分割为称为虚拟页(Virtual Page)的大小固定的块，每个虚拟页的大小为 P=pow(2,p)字节。类似的，物理内存被分割为物理页(Physical Page),大小也为 P 字节，物理页也被称为页帧。

虚拟内存的状态

- 未分配的：处于未分配状态的虚拟页是 VM 系统还未创建的页(严格来讲，他们此时并不能称之为虚拟页，它们是还未被纳入虚拟内存范围的磁盘块)。这些磁盘块没有任何数据与他们关联，因此它们不占用任何的磁盘空间。
- 缓存的：当前已缓存在物理内存中的已分配页。
- 未缓存的：当前未缓存在物理内存中的已分配页。



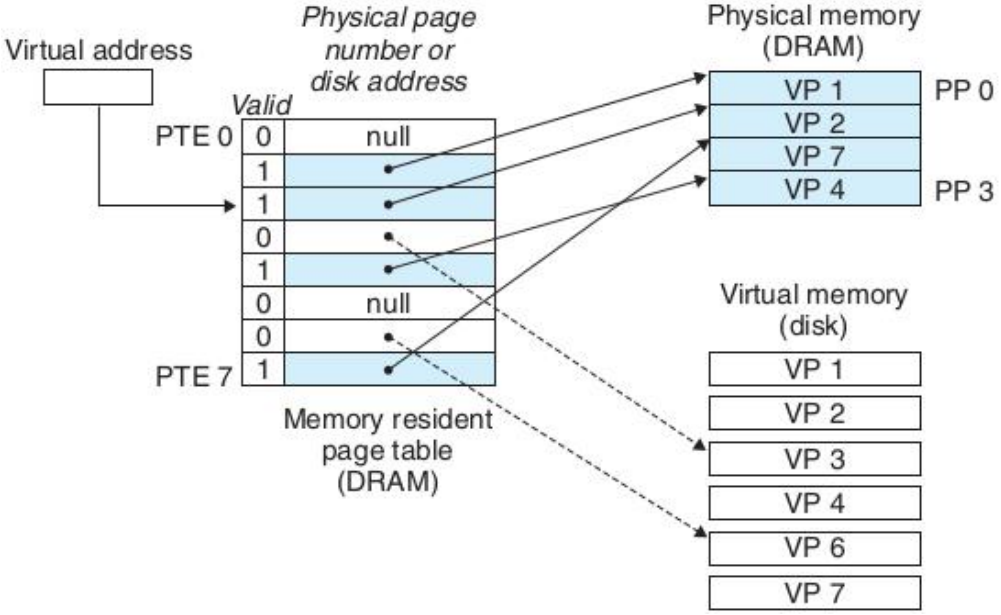
如上图展示了一个有 8 个虚拟页的虚拟内存，虚拟页 0,3 还未分配，因此在磁盘上还不存在，虚拟页 1,4,6 被缓存在物理内存中，页面 2,5,7 已经被分配了，但是当前并未被缓存在主存中。

页表

同任何缓存系统一样，虚拟内存系统必须有某种方法来判定一个虚拟页面是否缓存在物

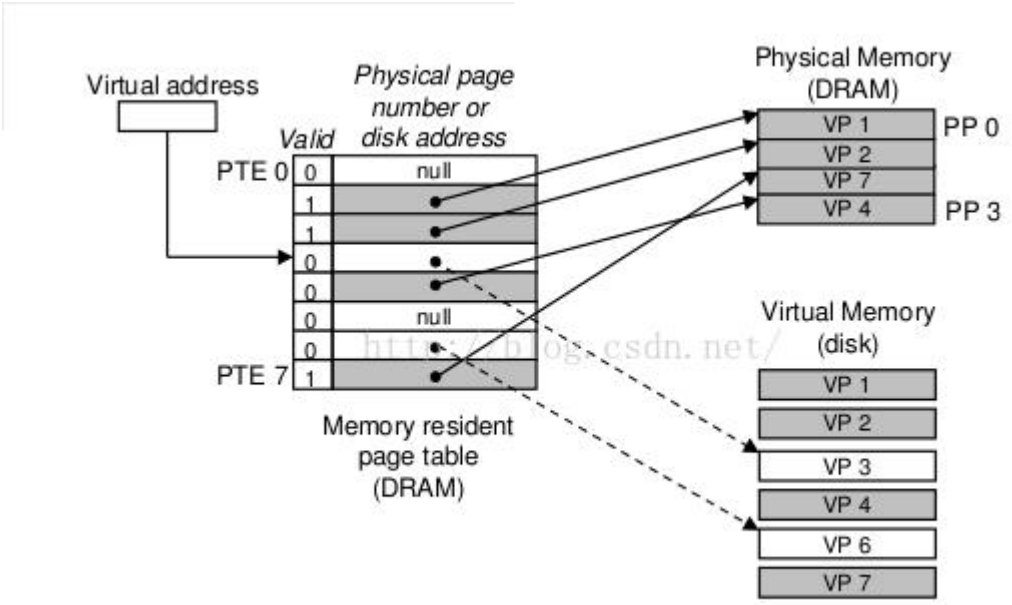
理内存中。如果是，系统还必须确定这个虚拟页存放在哪个物理页面中。如果不命中(**缺页**)，系统还必须确定这个虚拟页存放在磁盘的哪个位置，从而将该虚拟页面读入到物理内存中的某个物理页面中，如果物理内存中没有多余的物理页面，那么还将选择一个牺牲页将其移出(在这里移出的时候还要考虑该牺牲页是否已经更新，如果更新还要对应的更新虚拟页面里面的内容)，从而腾出空间放置新读入的物理页面的数据。

上述的这些功能是由软硬件联合提供的，包括操作系统软件、MMU 中的地址翻译硬件和一个存放在物理内存中的的叫做**页表**的数据结构。页表将虚拟页面映射到物理页面。每次地址翻译硬件将一个虚拟地址转换为物理地址的时候都要读取页表。

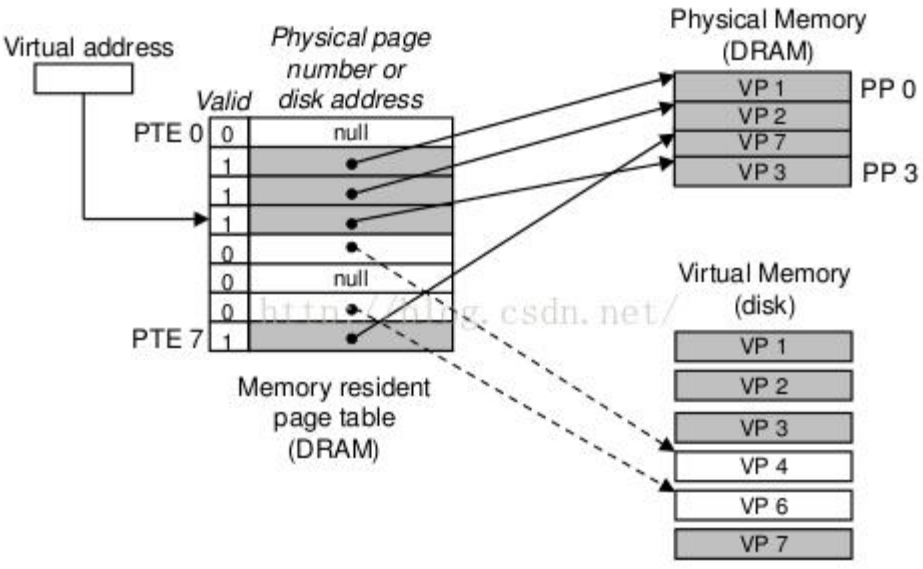


页命中与缺页

考虑一下当 CPU 想要读包含在 VP2 中的虚拟内存中的一个字时会发生什么，由于 VP2 已经被缓存在物理内存中。地址翻译硬件将虚拟地址作为索引来定位 PTE2,并且从物理内存中读取它(前面讲过页面位于物理内存中)，检查其有效位，发现其已经设置了有效位，此时地址翻译硬件就知道该页面已经缓存在物理内存中，并且该有效位后面的内容表示的就是对应的物理页面的地址。上面描述的这个过程就是页命中的情形。



下面我们来考虑一下缓存不命中的情形(在虚拟内存的习惯说法中，缓存不命中称为**缺页**)。CPU 想要读包含在 VP3 中的虚拟内存中的数据，地址翻译硬件根据 CPU 给出的虚拟地址，定位到页表中的 PTE3，检查发现有效位为 0，这表明当前 CPU 请求的页面并没有被缓存到物理内存中，这时就发生了缺页异常。这个缺页异常会调用内核中的**缺页异常处理程序**，该程序会选择一个牺牲页，在此例中该牺牲页就是存放在 PP3 中的 VP4。如果 VP4 已经被修改，那么内核就会把它赋值回磁盘。无论哪种情况，内核都会修改 VP4 中的页表条目，反映出 VP4 已经不再缓存在物理内存中这一事实。



接下来，内核从磁盘复制 VP3 到内存中的 PP3，更新 PTE3，此时我们的示例页表状态就成为上图这样子了。当异常处理程序返回时，该指令会把导致缺页的虚拟地址重新发送到地址翻译硬件。但是现在 VP3 已经缓存在主存中了，那么页面也就能够由地址翻译硬件正常处理了。

SWAP

前文提到了虚拟内存通过缺页中断为进程分配物理内存，内存总是有限的，如果所有的物理内存都被占用了怎么办呢？

Linux 提出 SWAP 的概念，Linux 中可以使用 SWAP 分区，在分配物理内存时，但可用内存不足时，将暂时不用的内存数据先放到磁盘上，让有需要的进程先使用，等进程再需要使用这些数据时，再将这些数据加载到内存中，通过这种"交换"技术，Linux 可以让进程使用更多的内存。

内存瓶颈的检测

Linux 内存的瓶颈，主要在于查看是否有比较严重的 swap 的发生(swap out)：

```
[root@db1 ~]# vmstat 2
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0  16640  21840      0 227016    0    0     4     3   57   75  0  1 99  0  0
 0  0  16640  21824      0 227004    0    0     0    27   69   91  1  1 99  0  0
 0  0  16640  21824      0 227004    0    0     0     4   73   93  0  1 99  0  0
 1  0  16640  21824      0 227004    0    0     0     0   74   92  0  1 99  0  0
 0  0  16640  21792      0 227004    0    0     0     0   68   86  0  1 99  0  0
 0  0  16640  21792      0 227004    0    0     0     0   69   91  0  1 99  0  0
```

si: Amount of memory swapped in from disk (/s).

so: Amount of memory swapped to disk (/s).

其他命令

top：

```
top - 03:15:10 up 3 days, 21:08,  2 users,  load average: 0.08, 0.04, 0.05
Tasks: 100 total,   1 running,  99 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.3 us,  1.7 sy,  0.0 ni, 97.6 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 st
KiB Mem :   499428 total,    29080 free,   240436 used,   229912 buff/cache
KiB Swap:  2097148 total,  2080508 free,    16640 used.   200780 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
25392 root      20   0       0      0      0 S  1.7  0.0   0:04.40 kworker/0:1
25479 root      20   0  161852   2224   1556 R  0.7  0.4   0:00.08 top
  457 root      20   0   34984   1864   1692 S  0.3  0.4   2:31.26 systemd-journal
  933 root      20   0  238984   3376   2080 S  0.3  0.7   2:09.67 rsyslogd
11433 mysql     20   0 1119012 174200  10952 S  0.3 34.9   3:02.37 mysqld
25391 root      20   0       0      0      0 S  0.3  0.0   0:03.74 kworker/0:0
25426 root      20   0  161348   6024   4640 S  0.3  1.2   0:00.16 sshd
25434 root      20   0  107380   5452   3468 S  0.3  1.1   0:00.02 dhclient
    1 root      20   0  125476   2688   1620 S  0.0  0.5   0:15.09 systemd
    2 root      20   0       0      0      0 S  0.0  0.0   0:00.07 kthreadd
    3 root      20   0       0      0      0 S  0.0  0.0   0:37.90 ksoftirqd/0
```

VITR：The total amount of virtual memory used by the task. It includes all code, data and shared libraries plus pages that have been swapped out and pages that have been mapped but not used.

RES：The non-swapped physical memory a task is using

free：free -m

pmap：pmap -x pid

压力测试

下面命令会减少可用内存的总量。它是通过在内存中创建文件系统然后往里面写文件来实现的。你可以使用任意多的内存，只需要往里面写入更多的文件就行了。

首先，创建一个挂载点，然后将 ramfs 文件系统挂载上去：

```
mkdir z
mount -t ramfs ramfs z/
```

第二步，使用 dd 在该目录下创建文件。这里我们创建了一个 128M 的文件

```
dd if=/dev/zero of=z/file bs=1M count=128
```

如何尽量避免 swap 对 mysql 的影响

- 尽量保证 Linux 操作系统还有足够的内存；
- 最新的内核，建议把 vm.swappiness 设置 1；
- 设置 /proc/$(pidof -s mysqld)/oom_adj 为较小的值来尽量避免 MySQL 由于系统内存不足而被关闭;
- 还可以在 mysqld 的配置文件 my.cnf 中[mysqld]段中加入 memlock=1 ，然后重启，避免 mysqld 发生 swap，但是可能会被 Linux oom kill 掉.

内核参数 vm.swappiness 的大小对如何使用 swap 分区有很大联系。值越大，表示越积极使用 swap 分区，越小表示越积极使用物理内存。默认值 swappiness=60，表示内存使用率超过 100-60=40%时开始使用交换分区。swappiness=0 的时候表示最大限度使用物理内存，然后才使用 swap 空间；swappiness = 100 的时候表示积极使用 swap 分区，并把内存上的数据及时搬运到 swap 空间。（对于 RedHat 2.6.32 之后的内核，设置为 0 会禁止使用 swap，可能会引发 out of memory，这种情况可以设置为 1。）

修改 Linux 内核参数 vm.swappiness

1、查看参数值：

```
cat /proc/sys/vm/swappiness
```

2、临时调整：

```
sysctl -w vm.swappiness=10
cat /proc/sys/vm/swappiness
```

3、永久调整：

```
vi /etc/sysctl.conf
vm.swappiness=10
```

4、然后加载参数：

```
sysctl -p
```

修改 oom_adj 值

1、查看 mysqld 的 oom_ajd 值

[root@localhost ~]# cat /proc/$(pidof -s mysqld)/oom_adj

0

默认值为 0. 当我们设置为-17 时，对于该进程来说，就不会触发 OOM 机制，被杀掉

2、修改：

[root@localhost ~]# echo -17 > /proc/$(pidof mysqld)/oom_adj

为什么是-17 呢？在 Linux 内核中的 oom.h 文件中，可以看到下面的定义：

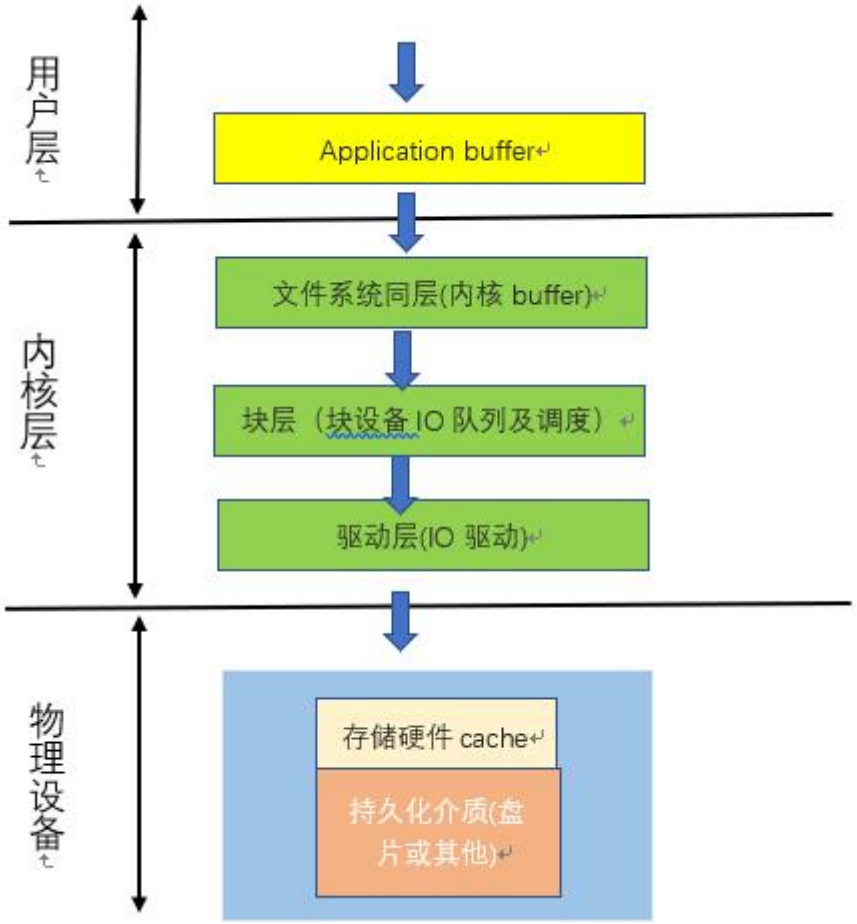
```
[root@db1 /]# cat /usr/include/linux/oom.h
.....
#define OOM_DISABLE (-17)
/* inclusive */
#define OOM_ADJUST_MIN (-16)
#define OOM_ADJUST_MAX 15

[root@db1 /]#
```

这个 oom_adj 中的变量的范围为 15 到-16 之间。越大越容易先被 kill。

Linux 磁盘 IO

IO 处理过程



IO 优化从以下几方面入手:

- 文件系统
  1. 文件系统类型
     在 rhel6.4 之前 ext4 性能比 xfs 好，因为 xfs 有 lock 争用的 bug。
     从 rhel6.4 开始，xfs 的 bug 被解决，xfs 性能比 ext4 好。
  2. 在挂载文件时添加参数：noatime，nodiratime
     默认方式下，在文件被访问、创建、修改等时候 linux 会记录下文件的一些时间戳(文件创建时间、最近一次修改时间和最近一次访问时间)。系统运行的时候要访问大量文件，如果能减少一些动作（比如减少时间戳的记录次数等）将会显著提高磁盘 IO 的效率。

mount –t ext4 –o rw,noatime ,nodiratime /dev/sda6 /data

- 通用块层

  该层就涉及到 IO 调度算法。系统中所有进程申请的 IO 操作, 全部在这里进行排队, 等待调度，然后写回磁盘。调度算法有四种：

  1. Anticipatory: 推迟 IO 请求(大约几个微秒)，以期能对他们进行排序，获得更高效率，适用于个人 PC 单磁盘系统；
  2. CFQ(Complete Fair Queuing) :完全公平的排队调度算法, 也就是每一个进程组之间按照公平的方式来调度 IO。显然适合多用户的系统，但是不适合作为数据库系统的 IO 调度算法。在数据库系统中，数据库进程肯定是 IO 最多的一个进程组，然后它却只能获得和其它进程一样多的 IO 调度机会，显然这是不合理的。数据库系统绝对不要使用该调度算法。
  3. Deadline: 按照截止期限来循环在各个 IO 队列进行调度，确保达到最终期限的请求被优先处理。一般 mysql 系统建议采用该调度算法。
  4. NOOP: 简单的 FIFO 队列进行调度, No operation 的意思是, 它没有进行额外的将临近的 IO 进行合并的操作, 所以它对 CPU 的使用极少。该调度算法特别适合于 SSD。因为 SSD 在对待顺序 IO 和随机 IO 没有什么区别。所以它不需要对临近的 IO 进行合并。避免了合并操作对 CPU 的使用。

- 磁盘

  对于随机读多的系统而言，磁盘很容易成为瓶颈所在，一般的优化就是使用 RAID 或者换 SSD

更改 I/O 调度方法

临时修改:

例如:想更改到 noop 电梯调度算法:

cat /sys/block/sda/queue/scheduler

echo noop > /sys/block/sda/queue/scheduler

永久修改

CentOS6：

vim /boot/grub/menu.lst

更改到如下内容:

kernel /boot/vmlinuz-2。6。32-504.el6 ro root=LABEL=/ elevator=deadline rhgb quiet

CentOS7：

[root@localhost ~]# grubby --update-kernel=ALL --args="elevator=deadline"

[root@localhost ~]# reboot

[root@localhost ~]# cat /sys/block/sda/queue/scheduler

noop [deadline] cfq

IO 瓶颈检测

- 使用 iostat 查看磁盘 IO

1. 显示单位： 默认 iostat 是以磁盘的 block 为单位，也可以使用 -k 来指定以 kilobytes 为单位，或者使用 -m 指定 megabytes 为单位；
2. 统计开始时间：默认 iostat 和 vmstat 相似，默认第一次/行都是从开机到目前的一个数据，可以使用 -y 选项去掉第一次/行的数据；
3. CPU 与磁盘： iostat 默认会显示 cpu 和磁盘的数据，如果只要 cpu 数据可以使用 -c 选项，如果只需要磁盘数据，可以使用 -d 选项；
4. 时间间隔和重复次数：[interval [times]] 表示磁盘统计数据的间隔时间和次数；

- iostat 指标解读

命令	指标	含义	提示
Iostat -c	iowait	等待 IO 完成时间百分比	
Iostat -d	tps	对磁盘每秒请求 IO 操作次数	
Iostat -x	rrqm/s	磁盘读时每秒发生多少次相邻磁盘的 merge 操作	
	wrqm/s	磁盘写时每秒发生多少次相邻磁盘的 merge 操作	
	r/s	每秒发送给磁盘的读请求数	之和为 IOPS
	w/s	每秒发送给磁盘的写请求数	
	rkB/s	每秒从磁盘读取的数据量	之和为吞吐量
	wkB/s	每秒从磁盘写入的数据量	
	avgrq-sz	平均一个 IO 请求涉及到多少个 sector	
	avgqu-sz	IO 队列的平均长度	
	await	平均每一个 IO 花费了多少毫秒（包括在 IO 队列中的排队时间和读写操作花费的时间）。这里可以理解为 IO 的响应时间，一般地系统 IO 响应时间应该低于 5ms，如果大于 10ms 就比较大了。	
	r_await	读请求处理完成等待时间	之和为响应时间
	w_await	写请求出来完成等待时间	
	%util	磁盘处理 I/O 的时间百分比	

- IO 瓶颈指标参考：

  CPU 的 %iowait io 等待很高；磁盘的 avgqu-sz 数值很大；await 数值很高；%util 数值很高；可能预示着磁盘存在瓶颈或者磁盘出现问题或者故障。

- 使用 iotop 找到 IO 最多的进程/线程
  a) 利用左右键 可以选择排序的字段，默认按照 IO>倒序，可以按照 SWAPIN，DISK WRITE 等等字段排序，使用左右方向键即可；
  b) 利用 p 键 可以在按照 进程显示 和按照 线程显示之间切换；
  c) 利用 r 键可以改变排序的方向：倒序 和 顺序

磁盘 IO 测试

eg：**stress** -d 1 --hdd-bytes 3G

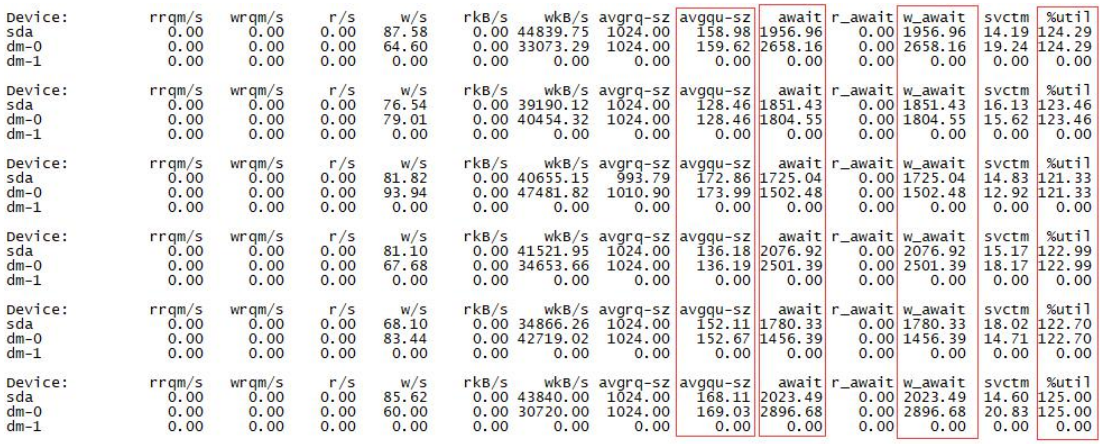
-d forks

--hdd forks 产生多个执行 write()函数的进程

--hdd-bytes bytes 指定写的 Bytes 数，默认是 1GB

--hdd-noclean 不要将写入随机 ASCII 数据的文件 Unlink(清除)

解释：-d 1：一个写进程。写入固定大小通过 mkstemp()函数写入当前目录；你也可以指定向磁盘中写入固定大小的文件，这个文件通过调用 mkstemp()产生并保存在当前目录下，默认是文件产生后就被执行 unlink（清除）操作，但是你可以使用“--hdd-noclean”选项将产生的文件全部保存在当前目录下，这会将你的磁盘空间逐步耗尽。



Device:	rrqm/s	wrqm/s	r/s	w/s	rkB/s	wkB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sda	0.00	0.00	0.00	87.58	0.00	44839.75	1024.00	158.98	1956.96	0.00	1956.96	14.19	124.29
dm-0	0.00	0.00	0.00	64.60	0.00	33073.29	1024.00	159.62	2658.16	0.00	2658.16	19.24	124.29
dm-1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Device:	rrqm/s	wrqm/s	r/s	w/s	rkB/s	wkB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sda	0.00	0.00	0.00	76.54	0.00	39190.12	1024.00	128.46	1851.43	0.00	1851.43	16.13	123.46
dm-0	0.00	0.00	0.00	79.01	0.00	40454.32	1024.00	128.46	1804.55	0.00	1804.55	15.62	123.46
dm-1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Device:	rrqm/s	wrqm/s	r/s	w/s	rkB/s	wkB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sda	0.00	0.00	0.00	81.82	0.00	40655.15	993.79	172.86	1725.04	0.00	1725.04	14.83	121.33
dm-0	0.00	0.00	0.00	93.94	0.00	47481.82	1010.90	173.99	1502.48	0.00	1502.48	12.92	121.33
dm-1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Device:	rrqm/s	wrqm/s	r/s	w/s	rkB/s	wkB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sda	0.00	0.00	0.00	81.10	0.00	41521.95	1024.00	136.18	2076.92	0.00	2076.92	15.17	122.99
dm-0	0.00	0.00	0.00	67.68	0.00	34653.66	1024.00	136.19	2501.39	0.00	2501.39	18.17	122.99
dm-1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Device:	rrqm/s	wrqm/s	r/s	w/s	rkB/s	wkB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sda	0.00	0.00	0.00	68.10	0.00	34866.26	1024.00	152.11	1780.33	0.00	1780.33	18.02	122.70
dm-0	0.00	0.00	0.00	83.44	0.00	42719.02	1024.00	152.67	1456.39	0.00	1456.39	14.71	122.70
dm-1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Device:	rrqm/s	wrqm/s	r/s	w/s	rkB/s	wkB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sda	0.00	0.00	0.00	85.62	0.00	43840.00	1024.00	168.11	2023.49	0.00	2023.49	14.60	125.00
dm-0	0.00	0.00	0.00	60.00	0.00	30720.00	1024.00	169.03	2896.68	0.00	2896.68	20.83	125.00
dm-1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Linux 网络管理

通过以下几方面来配置和优化网络：

1. TCP/IP 模型(了解)
2. TCP/IP 的三次握手建立连接和四次挥手结束连接
3. Socket Buffer
4. 多网卡绑定（了解）
5. 网络检测和故障排除

TCP/IP 模型

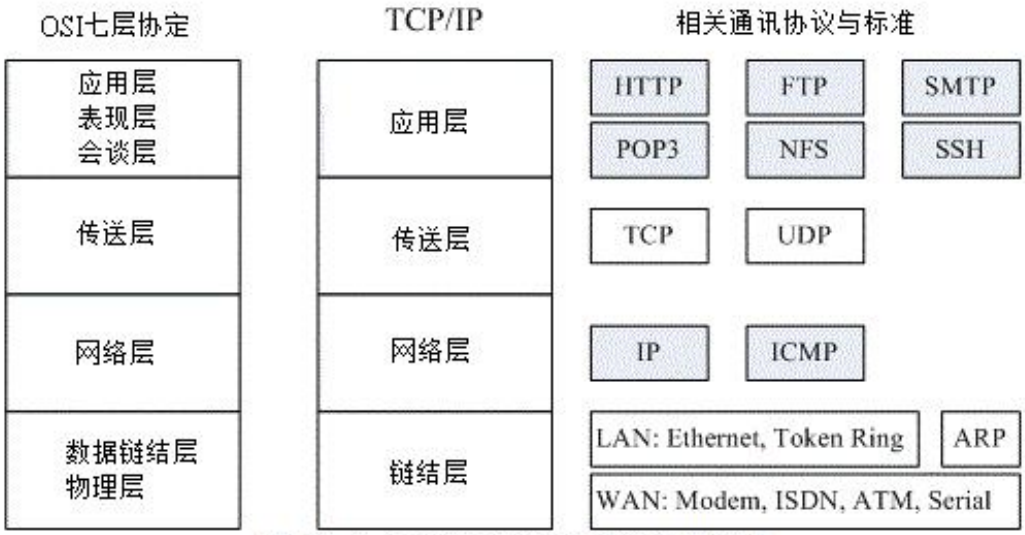
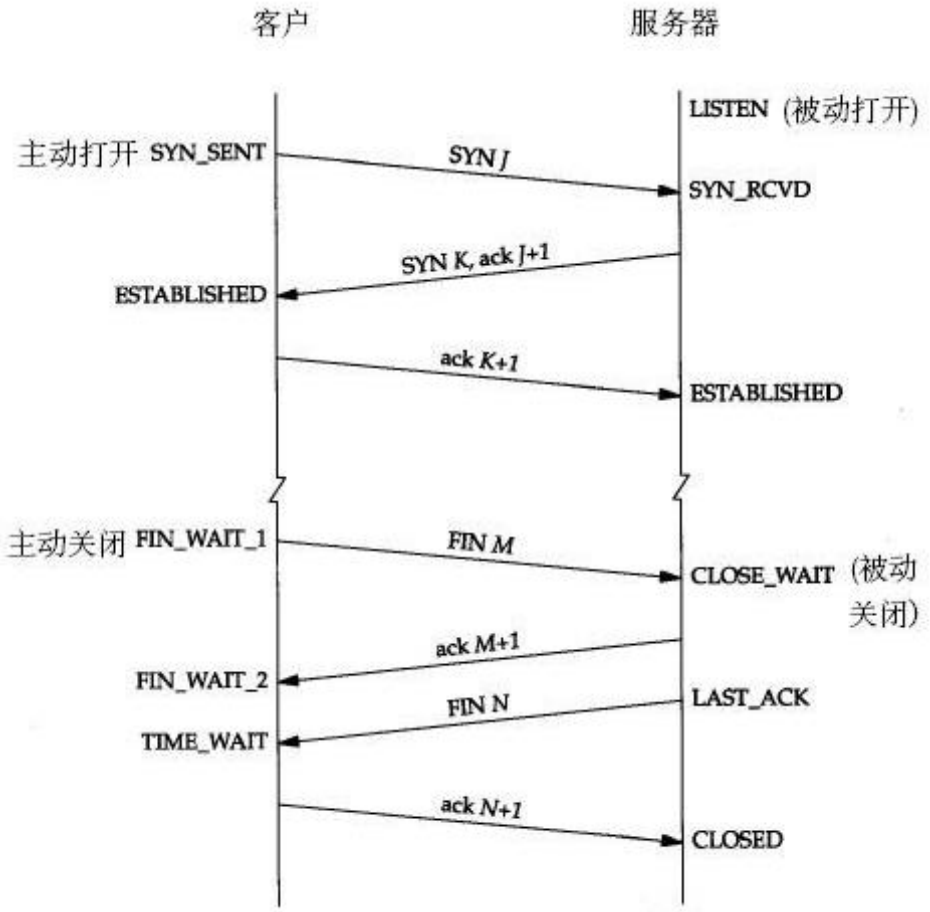


图 2.1-4、OSI 与 TCP/IP 协议之相关性

TCP/IP 的三次握手建立连接和四次挥手结束连接



三次握手建立连接：

1) 第一次握手 :建立连接时, 客户端 A 发送 SYN 包 (SYN=j) 到服务器 B, 并进入 SYN_SEND 状态，等待服务器 B 确认。
2) 第二次握手：服务器 B 收到 SYN 包，必须发送一个 ACK 包，来确认客户 A 的 SYN（ACK=j+1），同时自己也发送一个 SYN 包（SYN=k），即 SYN+ACK 包，此时服务器 B 进入 SYN_RECV 状态。
3) 第三次握手：客户端 A 收到服务器 B 的 SYN + ACK 包，向服务器 B 发送确认包 ACK（ACK=k+1），此包发送完毕，客户端 A 和服务器 B 进入 ESTABLISHED 状态，完成三次握手(注意, 主动打开方的最后一个 ACK 包中可能会携带了它要发送给服务端的数据)。

总结：三次握手，其实就是主动打开方，发送 SYN，表示要建立连接，然后被动打开方对此进行确认，然后主动方收到确认之后，对确认进行确认；

四次挥手断开连接：

由于 TCP 连接是全双工的，因此每个方向都必须单独进行关闭，TCP 的双方都要向对方发送一次 FIN 包，并且要对方对此进行确认。根据两次 FIN 包的发送和确认可以将四次挥手分为两个阶段：

- 第一阶段：主要是主动闭方发送 FIN，被动方对它进行确认；
  1) 第一次挥手：主动关闭方，客户端发送完数据之后，向服务器发送一个 FIN(M)数据包，进入 FIN_WAIT1 状态；被动关闭方服务器收到 FIN(M)后，进入 CLOSE_WAIT 状态；
  2) 第二次挥手：服务端发送 FIN(M)的确认包 ACK(M+1)，关闭服务器读通道，进入 LAST_ACK 状态；客户端收到 ACK(M+1)后，关闭客户端写通道，进入 FIN_WATI2 状态；此时客户端仍能通过读通道读取服务器的数据，服务器仍能通过写通道写数据。
- 第二阶段：主要是被动关闭方发送 FIN，主动方对它进行确认；
  1) 第三次挥手 :服务器发送完数据，向客户机发送一个 FIN(N)数据包，状态没有变还是 LAST_ACK；客户端收到 FIN(N)后，进入 TIME_WAIT 状态
  2) 第四次挥手：客户端返回对 FIN(N)的确认段 ACK(N+1)，关闭客户机读通道(还是 TIME_WAIT 状态);服务器收到 ACK(N+1)后，关闭服务器写通道，进入 CLOSED 状态。

总结：四次挥手，其本质就是：主动关闭方数据发送完成之后 发送 FIN，表示我方数据发送完，要断开连接，被动方对此进行确认；然后被动关闭方在数据发送完成之后 发送 FIN，表示我方数据发送完成，要断开连接，主动方对此进行确认；

CLOSE_WAIT 状态的原因和处理方法：

由 TCP 四次挥手断开连接的过程，可以知道 CLOSE_WAIT 是主动关闭方发送 FIN 之后，被动方收到 FIN 就进入了 CLOSE_WAIT 状态，此时如果被动方没有调用 close() 函数来关闭 TCP 连接，那么被动方服务器就会一直处于 CLOSE_WAIT 状态(等待调用 close 函数的状态)；所以 CLOSE_WAIT 状态的原因主要有两点：

1) 代码中没有写关闭连接的代码，也就是程序有 bug;
2) 该连接的业务代码处理时间太长，代码还在处理，对方已经发起断开连接请求；也就是客户端因为某种原因提前向服务端发出了 FIN 信号，导致服务端被动关闭，若服务端不主动关闭 socket 发 FIN 给 Client，此时服务端 Socket 会处于 CLOSE_WAIT 状态（而不是 LAST_ACK 状态）。

由于某种原因导致的 CLOSE_WAIT 会维持至少 2 个小时的时间（系统默认超时时间的是 7200 秒，也就是 2 小时）。如果服务端程序因某个原因导致系统造成过多 CLOSE_WAIT，那么通常是等不到释放那一刻系统就已崩掉了。要解决 CLOSE_WAIT 过多导致的问题，有两种方法：

1) 找到程序的 bug，进行修正；
2) 修改 TCP/IP 的 keepalive 的相关参数来缩短 CLOSE_WAIT 状态维持的时间

TCP 连接的保持(keepalive)相关参数：

1) /proc/sys/net/ipv4/tcp_keepalive_time 对应内核参数 net.ipv4.tcp_keepalive_time
   含义：如果在该参数指定的秒数内，TCP 连接一直处于空闲，则内核开始向客户端发起

对它的探测，看他是否还存活；

2) /proc/sys/net/ipv4/tcp_keepalive_intvl 对应内核参数 net.ipv4.tcp_keepalive_intvl
含义：以该参数指定的秒数为时间间隔，向客户端发起对它的探测；

3) /proc/sys/net/ipv4/tcp_keepalive_probes 对应内核参数 net.ipv4.tcp_keepalive_probes
含义：内核发起对客户端探测的次数，如果都没有得到响应，那么就断定客户端不可达或者已关闭，内核就关闭该 TCP 连接，释放相关资源

结论：CLOSE_WAIT 状态维持的秒数 = tcp_keepalive_time + tcp_keepalive_intvl * tcp_keepalive_probes。所以适当的降低 tcp_keepalive_time，tcp_keepalive_intvl，tcp_keepalive_probes 三个值就可以减少 CLOSE_WAIT：

临时修改方法：

sysctl -w net.ipv4.tcp_keepalive_time=600

sysctl -w net.ipv4.tcp_keepalive_probes=3

sysctl -w net.ipv4.tcp_keepalive_intvl=5

sysctl -p

修改会暂时生效,重新启动服务器后,会还原成默认值。修改之后，进行观察一段时间，如果 CLOSE_WAIT 减少，那么就可以进行永久性修改：

在文件 /etc/sysctl.conf 中的添加或者修改成下面的内容：

net.ipv4.tcp_keepalive_time = 1800

net.ipv4.tcp_keepalive_probes = 3

net.ipv4.tcp_keepalive_intvl = 15

这里的数值比上面的要大，因为上面的测试数据有点激进。当然数据该大之后效果不好，还是可以使用上面激进的数据。

修改之后执行：

sysctl -p

TIME_WAIT 状态的原因和处理方法：

TIME_WAIT 发生在 TCP 四次挥手的第二阶段：被动关闭方发送 FIN(N)，主动方收到该 FIN(N)，就进入 TIME_WAIT 状态，然后发送 ACK(N+1)。进入 TIME_WAIT 之后，主动关闭方会等待 2MSL(Maximum Segment Lifetime 报文最大存活时间)的时间，才释放自己占有的端口等资源。这是因为，如果最后的 ACK(N+1)没有被被动方收到的话，被动方会重新发生一个 FIN(N2)，那么主动方再次发送一个确认 ACK(N2+1)，所以这样一来就要用使主动关闭方在 TIME_WAIT 状态等待 2MSL 的时长。如果你的 TIME_WAIT 状态的 TCP 过多，占用了过多端口等资源，那么可以通过修改 TCP 内核参数进行调优：

TCP 连接的 TIME_WATI 相关参数：

1) /proc/sys/net/ipv4/tcp_tw_reuse 对应的内核参数：net.ipv4.tcp_tw_reuse
含义：是否能够重新启用处于 TIME_WAIT 状态的 TCP 连接用于新的连接；启用该参数的同时，必须同时启用下面的快速回收 recycle 参数

2) /proc/sys/net/ipv4/tcp_tw_recycle 对应的内核参数：net.ipv4.tcp_tw_recycle
含义：设置是否对 TIME_WAIT 状态的 TCP 进行快速回收；

3) /proc/sys/net/ipv4/tcp_fin_timeout 对应的内核参数：net.ipv4.tcp_fin_timeout
含义：主动关闭方 TCP 保持在 FIN_WAIT_2 状态的时间。对方可能会一直不结束连接或不可预料的进程死亡。默认值为 60 秒。

```
修改方法和 keepalive 的相关参数一样：
sysctl -w net.ipv4.tcp_tw_reuse=1
sysctl -w net.ipv4.tcp_tw_recycle=1
sysctl -w net.ipv4.tcp_fin_timeout=30
sysctl -p

永久修改方法，也是修改 /etc/sydctl.conf:
net.ipv4.tcp_tw_reuse=1
net.ipv4.tcp_tw_recycle=1
net.ipv4.tcp_fin_timeout=30
sysctl -p 是修改生效。
```

Socket Buffer

发送方发送数据，接收方接受数据，双方必须存在一个保存数据的 buffer，称为 Socket Buffer，TCP/IP 的实现都是放在 kernel 中的，所以 Socket Buffer 也是在 kernel 中的。Socket Buffer 的大小配置对网络的性能有很大的影响，相关参数如下：

1) /proc/sys/net/ipv4/tcp_mem: 这是一个系统全局参数，表示所有 TCP 的 buffer 配置。有三个值，单位为内存页(通常为 4K)，第一个值表示 buffer 的下限，第二个值表示内存压力模式开启，即超过该值，开启内存压力模式；第三个值内存使用的上限，超过时，可能会丢弃报文。
2) /proc/sys/net/ipv4/tcp_rmem: r 表示 receive，也有三个值，第一个值为 TCP 接收 buffer 的最少字节数；第二个是默认值(该值会被 rmem_default 覆盖)；第三个值 TCP 接收 buffer 的最大字节数(该值会被 rmem_max 覆盖)；
3) /proc/sys/net/ipv4/tcp_wmem: w 表示 write，也就是 send。也有三个值，第一个值为 TCP 发送 buffer 的最少字节数 ;第二个是默认值(该值会被 wmem_default 覆盖) ;第三个值 TCP 发送 buffer 的最大字节数(该值会被 wmem_max 覆盖)；
4) /proc/sys/net/core/wmem_default: TCP 数据发送窗口默认字节数；
5) /proc/sys/net/core/wmem_max: TCP 数据发送窗口最大字节数；
6) /proc/sys/net/core/rmem_default: TCP 数据接收窗口默认字节数；
7) /proc/sys/net/core/rmem_max: TCP 数据接收窗口最大字节数；

调高网络缓存

```
/proc/sys/net/ipv4/tcp_mem TCP 全局缓存，单位为内存页(4k);
对应的内核参数：net.ipv4.tcp_mem ,可以在 /etc/sysctl.conf 中进行修改；
/proc/sys/net/ipv4/tcp_rmem 接收 buffer，单位为字节
对应的内核参数：net.ipv4.tcp_rmem，可以在 /etc/sysctl.conf 中进行修改；
/proc/sys/net/ipv4/tcp_wmem 接收 buffer，单位为字节
对应的内核参数：net.ipv4.tcp_wmem，可以在 /etc/sysctl.conf 中进行修改；
```

/proc/sys/net/core/rmem_default 接收 buffer 默认大小，单位字节

对应内核参数：net.core.rmem_default，可以在 /etc/sysctl.conf 中进行修改；

/proc/sys/net/core/rmem_max 接收 buffer 最大大小，单位字节

对应内核参数：net.core.rmem_max，可以在 /etc/sysctl.conf 中进行修改；

/proc/sys/net/core/wmem_default 发送 buffer 默认大小，单位字节

对应内核参数：net.core.rmem_default，可以在 /etc/sysctl.conf 中进行修改；

/proc/sys/net/core/wmem_max 发送 buffer 最大大小，单位字节

对应内核参数：net.core.rmem_max，可以在 /etc/sysctl.conf 中进行修改；

注意：修改 /etc/sysctl.conf 之后，必须执行 sysctl -p 命令才能生效。

多网卡绑定(了解)

Linux 内核支持将多个物理网卡绑定成一个逻辑网络，通过 bond 技术让多块网卡看起来是一个单独的以太网接口设备并具有相同的 ip 地址，从而进行网卡的负载均衡和网卡容错。

网络检测和故障排除

1. 延时检测、端口及防火墙的状态检测
   ping 、telnet、systemctl status firewalld(centos 7.0)

2. 使用 netstat 命令查看活动的网络连接

根据协议 tcp -t, udp -u 进行显示： netstat -ntap ， netstat -nuap

-n 表示 numeric 用数字显示 ip 和 端口，而不是文字；

-t 表示 tcp 连接

-u 表示 udp 连接

-a 选项表示所有状态的 TCP 连接

-p 表示程序名称；

字段含义：

recv-Q 接收队列(receive queue)。表示收到的数据已经在本地接收缓冲，但是还有多少没有被用户进程取走。

send-Q 发送队列(send queue)。发送了数据，但是没有收到对方的 Ack 的，还保留本地缓冲区。

注意：这两个值通常应该为 0，如果不为 0 可能是有问题的。packets 在两个队列里都不应该有堆积状态。可接受短暂的非 0 情况。

查看 TCP 连接处于各种状态的连接数量：

[root@db1 ~]# netstat -n | awk '/^tcp/ {++S[$NF]} END {for(a in S) print a, S[a]}'

ESTABLISHED 4

3. 使用 sar 命令查看网络流量及负载

sar -n DEV 1 5：查看所有网卡每秒收包，发包数量，每秒接收多少 KB，发送多少 KB：

字段含义：rxpck/s txpck/s rxkB/s txkB/s 每秒收包，每秒发包，每秒接收 KB 数量，每秒发送 KB 数量；

sar -n EDEV 1 5：查看网络错误，网络超负载，网络故障

字段含义：

rxerr/s：Total number of bad packets received per second. 每秒接收的损坏的包

txerr/s：Total number of errors that happened per second while transmitting packets. 每秒发送的损坏的包

coll/s: Number of collisions that happened per second while transmitting packets. 每秒发送的网络冲突包数

rxdrop/s： Number of received packets dropped per second because of a lack of space in linux buffers. 每秒丢弃的接收包

txdrop/s： Number of transmitted packets dropped per second because of a lack of space in linux buffers. 每秒发送的被丢弃的包

txcarr/s： Number of carrier-errors that happened per second while transmitting packets. carrier-errors 每秒载波错误数

rxfram/s： Number of frame alignment errors that happened per second on received packets.每秒接收数据包的帧对齐错误数

rxfifo/s：Number of FIFO overrun errors that happened per second on received packets. 接收 fifo 队列发生过载错误次数(每秒)

txfifo/s：Number of FIFO overrun errors that happened per second on transmitted packets. 发送方的 fifo 队列发生过载错误次数(每秒)

1) 如果 coll/s 网络冲突持续存在，那么可能网络设备存在问题或者存在瓶颈，或者配置错误；

2) 发生了 FIFO 队列 overrun，表示 SOCKET BUFFER 太小；

3) 持续的数据包损坏，frame 损坏，可能预示着网络设备出现部分故障，或者配置错误；

4. 使用 ifconfig 命令查看网络活动

```
[root@localhost ~]# ifconfig eth0
// UP：表示“接口已启用”。
// BROADCAST ：表示“主机支持广播”。
// RUNNING：表示“接口在工作中”。
// MULTICAST：表示“主机支持多播”。
// MTU:1500（最大传输单元）：1500 字节
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500

// inet ：网卡的 IP 地址。
// netmask ：网络掩码。
// broadcast ：广播地址。
inet 192.168.1.135 netmask 255.255.255.0 broadcast 192.168.1.255

// 网卡的 IPv6 地址
inet6 fe80::2aa:bbff:fecc:ddee prefixlen 64 scopeid 0x20<link>

// 连接类型：Ethernet (以太网) HWaddr (硬件 mac 地址)
// txqueuelen (网卡设置的传送队列长度)
ether 00:aa:bb:cc:dd:ee txqueuelen 1000 (Ethernet)

// RX packets 接收时，正确的数据包数。
// RX bytes 接收的数据量。
// RX errors 接收时，产生错误的数据包数。
// RX dropped 接收时，丢弃的数据包数。
```

// RX overruns 接收时，由于速度过快而丢失的数据包数。

// RX frame 接收时，发生 frame 错误而丢失的数据包数。

RX packets 2825 bytes 218511 (213.3 KiB)

RX errors 0 dropped 0 overruns 0 frame 0

// TX packets 发送时，正确的数据包数。

// TX bytes 发送的数据量。

// TX errors 发送时，产生错误的数据包数。

// TX dropped 发送时，丢弃的数据包数。

// TX overruns 发送时，由于速度过快而丢失的数据包数。

// TX carrier 发送时，发生 carrier 错误而丢失的数据包数。

// collisions 冲突信息包的数目。

TX packets 1077 bytes 145236 (141.8 KiB)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0