**管理处理器的亲和性（affinity）**

为什么（3 个原因）以及如何使用硬（相对于软）CPU 亲和性（affinity）

# 了解 Linux® 2.6 调度器如何处理 CPU 亲和性（affinity）可以帮助您更好地设计用户空间的应用程序。软亲和性（affinity） 意味着进程并不会在处理器之间频繁迁移，而 硬亲和性（affinity） 则意味着进程需要在您指定的处理器上运行。本文介绍了当前的亲和性（affinity）机制，解释为什么和如何使用亲和性（affinity），并给出了几个样例代码来显示如何使用这种功能。

# 简单地说，CPU 亲和性（affinity） 就是进程要在某个给定的 CPU 上尽量长时间地运行而不被迁移到其他处理器的倾向性。Linux 内核进程调度器天生就具有被称为 软 CPU 亲和性（affinity） 的特性，这意味着进程通常不会在处理器之间频繁迁移。这种状态正是我们希望的，因为进程迁移的频率小就意味着产生的负载小。

# 2.6 版本的 Linux 内核还包含了一种机制，它让开发人员可以编程实现 硬 CPU 亲和性（affinity）。这意味着应用程序可以显式地指定进程在哪个（或哪些）处理器上运行。

# 什么是 Linux 内核硬亲和性（affinity）？

# 在 Linux 内核中，所有的进程都有一个相关的数据结构，称为 task\_struct。这个结构非常重要，原因有很多；其中与 亲和性（affinity）相关度最高的是 cpus\_allowed 位掩码。这个位掩码由 n 位组成，与系统中的 n 个逻辑处理器一一对应。具有 4 个物理 CPU 的系统可以有 4 位。如果这些 CPU 都启用了超线程，那么这个系统就有一个 8 位的位掩码。

# 如果为给定的进程设置了给定的位，那么这个进程就可以在相关的 CPU 上运行。因此，如果一个进程可以在任何 CPU 上运行，并且能够根据需要在处理器之间进行迁移，那么位掩码就全是 1。实际上，这就是 Linux 中进程的缺省状态。

Linux 内核 API 提供了一些方法，让用户可以修改位掩码或查看当前的位掩码：

* sched\_set\_affinity() （用来修改位掩码）
* sched\_get\_affinity() （用来查看当前的位掩码）

注意，cpu\_affinity 会被传递给子线程，因此应该适当地调用 sched\_set\_affinity。

**为什么应该使用硬亲和性（affinity）？**

通常 Linux 内核都可以很好地对进程进行调度，在应该运行的地方运行进程（这就是说，在可用的处理器上运行并获得很好的整体性能）。内核包含了一些用来检测 CPU 之间任务负载迁移的算法，可以启用进程迁移来降低繁忙的处理器的压力。

一般情况下，在应用程序中只需使用缺省的调度器行为。然而，您可能会希望修改这些缺省行为以实现性能的优化。让我们来看一下使用硬亲和性（affinity） 的 3 个原因。

**原因 1. 有大量计算要做**

基于大量计算的情形通常出现在科学和理论计算中，但是通用领域的计算也可能出现这种情况。一个常见的标志是您发现自己的应用程序要在多处理器的机器上花费大量的计算时间。

**原因 2. 您在测试复杂的应用程序**

测试复杂软件是我们对内核的亲和性（affinity）技术感兴趣的另外一个原因。考虑一个需要进行线性可伸缩性测试的应用程序。有些产品声明可以在 *使用更多硬件* 时执行得更好。

我们不用购买多台机器（为每种处理器配置都购买一台机器），而是可以：

* 购买一台多处理器的机器
* 不断增加分配的处理器
* 测量每秒的事务数
* 评估结果的可伸缩性

如果应用程序随着 CPU 的增加可以线性地伸缩，那么每秒事务数和 CPU 个数之间应该会是线性的关系（例如斜线图 —— 请参阅下一节的内容）。这样建模可以确定应用程序是否可以有效地使用底层硬件。

Amdahl 法则说明这种加速比在现实中可能并不会发生，但是可以非常接近于该值。对于通常情况来说，我们可以推论出每个程序都有一些串行的组件。随着问题集不断变大，串行组件最终会在优化解决方案时间方面达到一个上限。

Amdahl 法则在希望保持高 CPU 缓存命中率时尤其重要。如果一个给定的进程迁移到其他地方去了，那么它就失去了利用 CPU 缓存的优势。实际上，如果正在使用的 CPU 需要为自己缓存一些特殊的数据，那么所有其他 CPU 都会使这些数据在自己的缓存中失效。

因此，如果有多个线程都需要相同的数据，那么将这些线程绑定到一个特定的 CPU 上是非常有意义的，这样就确保它们可以访问相同的缓存数据（或者至少可以提高缓存的命中率）。否则，这些线程可能会在不同的 CPU 上执行，这样会频繁地使其他缓存项失效。

**原因 3. 您正在运行时间敏感的、决定性的进程**

我们对 CPU 亲和性（affinity）感兴趣的最后一个原因是实时（对时间敏感的）进程。例如，您可能会希望使用硬亲和性（affinity）来指定一个 8 路主机上的某个处理器，而同时允许其他 7 个处理器处理所有普通的系统调度。这种做法确保长时间运行、对时间敏感的应用程序可以得到运行，同时可以允许其他应用程序独占其余的计算资源。

下面的样例应用程序显示了这是如何工作的。

**如何利用硬亲和性（affinity）**

现在让我们来设计一个程序，它可以让 Linux 系统非常繁忙。可以使用前面介绍的系统调用和另外一些用来说明系统中有多少处理器的 API 来构建这个应用程序。实际上，我们的目标是编写这样一个程序：它可以让系统中的每个处理器都繁忙几秒钟。可以从后面的“下载”一节中 [下载样例程序](http://www.ibm.com/developerworks/cn/linux/l-affinity.html#download)。

Amdahl 法则

Amdahl 法则是有关使用并行处理器来解决问题相对于只使用一个串行处理器来解决问题的加速比的法则。加速比（Speedup） 等于串行执行（只使用一个处理器）的时间除以程序并行执行（使用多个处理器）的时间：

S =T(1)/I(j)

其中 T(j) 是在使用 j 个处理器执行程序时所花费的时间。

**清单 1. 让处理器繁忙**

/\* This method will create threads, then bind each to its own cpu. \*/

bool do\_cpu\_stress(int numthreads)

{

int ret = TRUE;

int created\_thread = 0;

/\* We need a thread for each cpu we have... \*/

while ( created\_thread < numthreads - 1 )

{

int mypid = fork();

if (mypid == 0) /\* Child process \*/

{

printf("\tCreating Child Thread: #%i\n", created\_thread);

break;

}

else /\* Only parent executes this \*/

{

/\* Continue looping until we spawned enough threads! \*/ ;

created\_thread++;

}

}

/\* NOTE: All threads execute code from here down! \*/

正如您可以看到的一样，这段代码只是通过 fork 调用简单地创建一组线程。每个线程都执行这个方法中后面的代码。现在我们让每个线程都将亲和性（affinity）设置为自己的 CPU。

**清单 2. 为每个线程设置 CPU 亲和性（affinity）**

cpu\_set\_t mask;

/\* CPU\_ZERO initializes all the bits in the mask to zero. \*/

CPU\_ZERO( &mask );

/\* CPU\_SET sets only the bit corresponding to cpu. \*/

CPU\_SET( created\_thread, &mask );

/\* sched\_setaffinity returns 0 in success \*/

if( sched\_setaffinity( 0, sizeof(mask), &mask ) == -1 )

{

printf("WARNING: Could not set CPU Affinity, continuing...\n");

}

如果程序可以执行到这儿，那么我们的线程就已经设置了自己的亲和性（affinity）。调用 sched\_setaffinity 会设置由 pid 所引用的进程的 CPU 亲和性（affinity）掩码。如果 pid 为 0，那么就使用当前进程。

亲和性（affinity）掩码是使用在 mask 中存储的位掩码来表示的。最低位对应于系统中的第一个逻辑处理器，而最高位则对应于系统中最后一个逻辑处理器。

每个设置的位都对应一个可以合法调度的 CPU，而未设置的位则对应一个不可调度的 CPU。换而言之，*进程都被绑定了，只能在那些对应位被设置了的处理器上运行*。通常，掩码中的所有位都被置位了。这些线程的亲和性（affinity）都会传递给从它们派生的子进程中。

注意不应该直接修改位掩码。应该使用下面的宏。虽然在我们的例子中并没有全部使用这些宏，但是在本文中还是详细列出了这些宏，您在自己的程序中可能需要这些宏。

**清单 3. 间接修改位掩码的宏**

void CPU\_ZERO (cpu\_set\_t \*set)

这个宏对 CPU 集 set 进行初始化，将其设置为空集。

void CPU\_SET (int cpu, cpu\_set\_t \*set)

这个宏将 cpu 加入 CPU 集 set 中。

void CPU\_CLR (int cpu, cpu\_set\_t \*set)

这个宏将 cpu 从 CPU 集 set 中删除。

int CPU\_ISSET (int cpu, const cpu\_set\_t \*set)

如果 cpu 是 CPU 集 set 的一员，这个宏就返回一个非零值（true），否则就返回零（false）。

对于本文来说，样例代码会继续让每个线程都执行某些计算量较大的操作。

**清单 4. 每个线程都执行一个计算敏感的操作**

/\* Now we have a single thread bound to each cpu on the system \*/

int computation\_res = do\_cpu\_expensive\_op(41);

cpu\_set\_t mycpuid;

sched\_getaffinity(0, sizeof(mycpuid), &mycpuid);

if ( check\_cpu\_expensive\_op(computation\_res) )

{

printf("SUCCESS: Thread completed, and PASSED integrity check!\n",

mycpuid);

ret = TRUE;

}

else

{

printf("FAILURE: Thread failed integrity check!\n",

mycpuid);

ret = FALSE;

}

return ret;

}

现在您已经了解了在 Linux 2.6 版本的内核中设置 CPU 亲和性（affinity）的基本知识。接下来，我们使用一个 main 程序来封装这些方法，它使用一个用户指定的参数来说明要让多少个 CPU 繁忙。我们可以使用另外一个方法来确定系统中有多少个处理器：

int NUM\_PROCS = sysconf(\_SC\_NPROCESSORS\_CONF);

这个方法让程序能够自己确定要让多少个处理器保持繁忙，例如缺省让所有的处理器都处于繁忙状态，并允许用户指定系统中实际处理器范围的一个子集。

**运行样例程序**

当运行前面介绍的 [样例程序](http://www.ibm.com/developerworks/cn/linux/l-affinity.html#download) 时，可以使用很多工具来查看 CPU 是否是繁忙的。如果只是简单地进行测试，可以使用 Linux 命令 top。在运行 top 命令时按下 “1” 键，可以看到每个 CPU 执行进程所占用的百分比。

**结束语**

这个样例程序虽然非常简单，但是它却展示了使用 Linux 内核中实现的硬亲和性（affinity）的基本知识。（任何使用这段代码的应用程序都无疑会做一些更有意义的事情。）了解了 CPU 亲和性（affinity）内核 API 的基本知识，您就可以从复杂的应用程序中榨取出最后一点儿性能了。

# 了解 Linux® 2.6 调度器如何处理 CPU 亲和性（affinity）可以帮助您更好地设计用户空间的应用程序。软亲和性（affinity） 意味着进程并不会在处理器之间频繁迁移，而 硬亲和性（affinity） 则意味着进程需要在您指定的处理器上运行。本文介绍了当前的亲和性（affinity）机制，解释为什么和如何使用亲和性（affinity），并给出了几个样例代码来显示如何使用这种功能。了解 Linux® 2.6 调度器如何处理 CPU 亲和性（affinity）可以帮助您更好地设计用户空间的应用程序。软亲和性（affinity） 意味着进程并不会在处理器之间频繁迁移，而 硬亲和性（affinity） 则意味着进程需要在您指定的处理器上运行。本文介绍了当前的亲和性（affinity）机制，解释为什么和如何使用亲和性（affinity），并给出了几个样例代码来显示如何使用这种功能。管理处理器的亲和性（affinity）管理处理器的亲和性（affinity）

为什么（3 个原因）以及如何使用硬（相对于软）CPU 亲和性（affinity）

了解 Linux® 2.6 调度器如何处理 CPU 亲和性（affinity）可以帮助您更好地设计用户空间的应用程序。软亲和性（affinity） 意味着进程并不会在处理器之间频繁迁移，而 硬亲和性（affinity） 则意味着进程需要在您指定的处理器上运行。本文介绍了当前的亲和性（affinity）机制，解释为什么和如何使用亲和性（affinity），并给出了几个样例代码来显示如何使用这种功能。

管理处理器的亲和性（affinity）

*为什么（3 个原因）以及如何使用硬（相对于软）CPU 亲和性（affinity）*

了解 Linux® 2.6 调度器如何处理 CPU 亲和性（affinity）可以帮助您更好地设计用户空间的应用程序。*软亲和性（affinity）* 意味着进程并不会在处理器之间频繁迁移，而 *硬亲和性（affinity）* 则意味着进程需要在您指定的处理器上运行。本文介绍了当前的亲和性（affinity）机制，解释为什么和如何使用亲和性（affinity），并给出了几个样例代码来显示如何使用这种功能。

# 管理处理器的亲和性（affinity）

为什么（3 个原因）以及如何使用硬（相对于软）CPU 亲和性（affinity）

了解 Linux® 2.6 调度器如何处理 CPU 亲和性（affinity）可以帮助您更好地设计用户空间的应用程序。软亲和性（affinity） 意味着进程并不会在处理器之间频繁迁移，而 硬亲和性（affinity） 则意味着进程需要在您指定的处理器上运行。本文介绍了当前的亲和性（affinity）机制，解释为什么和如何使用亲和性（affinity），并给出了几个样例代码来显示如何使用这种功能。