# 一．多线程属性pthread\_attr详解

Posix线程中的线程属性pthread\_attr\_t主要包括scope属性、detach属性、堆栈地址、堆栈大小、优先级。在pthread\_create中，把第二个参数设置为NULL的话，将采用默认的属性配置。

线程属性结构如下：

typedef struct

{

int detachstate; 线程的分离状态

int schedpolicy; 线程调度策略

struct sched\_param schedparam; 线程的调度参数

int inheritsched; 线程的继承性

int scope; 线程的作用域

size\_t guardsize; 线程栈末尾的警戒缓冲区大小

int stackaddr\_set;

void \* stackaddr; 线程栈的位置

size\_t stacksize; 线程栈的大小

}pthread\_attr\_t;

**detachstate**，表示新线程是否与进程中其他线程脱离同步， 如果设置为PTHREAD\_CREATE\_DETACHED 则新线程不能用pthread\_join()来同步，且在退出时自行释放所占用的资源。

缺省为PTHREAD\_CREATE\_JOINABLE状态。这个属性也可以在线程创建并运行以后用pthread\_detach()来设置，而一旦设置为PTHREAD\_CREATE\_DETACH状态（不论是创建时设置还是运行时设置）则不能再恢复到PTHREAD\_CREATE\_JOINABLE状态。

线程的分离状态决定一个线程以什么样的方式来终止自己。在默认情况下线程是非分离状态的，这种情况下，原有的线程等待创建的线程结束。只有当pthread\_join() 函数返回时，创建的线程才算终止，才能释放自己占用的系统资源。分离线程没有被其他的线程所等待，自己运行结束了，线程也就终止了，马上释放系统资源。

通俗的说也就是：我们知道一般我们要等待(pthread\_join)一个线程的结束，主要是想知道它的结束状态，否则等待一般是没有什么意义的！但是if有一些线程的终止态我们压根就不想知道，那么就可以使用“分离”属性，那么我 们就无须等待管理，只要线程自己结束了，自己释放src就可以咯

#include <pthread.h>

int pthread\_attr\_getdetachstate(const pthread\_attr\_t \* attr, int \* detachstate);

int pthread\_attr\_setdetachstate(pthread\_attr\_t \* attr, int detachstate);

参数：

attr:线程属性变量

detachstate:分离状态属性  
若成功返回0，若失败返回-1。  
设置的时候可以有两种选择：  
<1>.detachstate参数为：PTHREAD\_CREATE\_DETACHED 分离状态启动  
<2>.detachstate参数为：PTHREAD\_CREATE\_JOINABLE 正常启动线程

**\_\_schedpolicy**，表示新线程的调度策略，主要包括：  
SCHED\_OTHER（正常、非实时）  
SCHED\_RR（实时、轮转法）  
SCHED\_FIFO（实时、先入先出）  
缺省为SCHED\_OTHER，后两种调度策略仅对超级用户有效。

运行时可以用过pthread\_setschedparam()来改变。

函数pthread\_attr\_setschedpolicy和pthread\_attr\_getschedpolicy分别用来设置和得到线程的调度策略。

int pthread\_attr\_getschedpolicy(const pthread\_attr\_t \*, int \* policy)

int pthread\_attr\_setschedpolicy(pthread\_attr\_\*, int policy)

参数：  
attr 线程属性变量  
policy 调度策略  
若成功返回0，若失败返回-1。

所谓调度策略也就是我们之前在OS中所学过的那些调度算法：  
SCHED\_FIFO ：先进先出  
SCHED\_RR ：轮转法  
SCHED\_OTHER ：其他方法  
SCHED\_OTHER是不支持优先级使用的,而SCHED\_FIFO和SCHED\_RR  
支持优先级的使用,他们分别为1和99,数值越大优先级越高.

注意：  
> 此处的SCHED\_FIFO是允许被高优先级抢占的！  
> 也就是有高优先级的必须先运行  
> SCHED\_RR是设置一个时间片  
> 当有SCHED\_FIFO或SCHED\_RR策赂的线程在一个条件变量  
上等持或等持加锁同一个互斥量时，它们将以优先级顺序被唤  
醒。即，如果一个低优先级的SCHED\_FIFO线程和一个高优先  
织的SCHED\_FIFO线程都在等待锁相同的互斥且，则当互斥量  
被解锁时，高优先级线程将总是被首先解除阻塞。

**\_\_schedparam**，一个struct sched\_param结构，目前仅有一个sched\_priority整型变量表示线程的运行优先级。这个参数仅当调度策略为实时（即SCHED\_RR或SCHED\_FIFO）时才有效，并可以在运行时通过pthread\_setschedparam()函数来改变，缺省为0。  
函数pthread\_attr\_getschedparam 和pthread\_attr\_setschedparam分别用来设置和得到线程的调度参数。

int pthread\_attr\_getschedparam(const pthread\_attr\_t \*,struct sched\_param \*);

int pthread\_attr\_setschedparam(pthread\_attr\_t \*,const struct sched\_param \*);

参数：  
attr 线程变量属性  
param sched\_parm 结构体  
若成功返回0，若失败返回-1。  
/usr/include /bits/sched.h  
struct sched\_param  
{  
int sched\_priority; // 参数的本质就是优先级  
};  
注意：大的权值对应高的优先级!  
系统支持的最大和最小的优先级值可以用函数：  
sched\_get\_priority\_max和sched\_get\_priority\_min得到！

#include <pthread.h>

int sched\_get\_priority\_max( int policy );

int sched\_get\_priority\_min( int policy );

参数：

max\_： 系统支持的优先级的最小值  
min\_ ： 系统支持的优先级的最大值

max\_ = sched\_get\_priority\_max( policy );

min\_ = sched\_get\_priority\_min( policy );

注意参数是policy调用策略，也就是说对于不同的策略的值是不  
一样的！  
policy = SCHED\_OTHER  
max\_priority = 0  
min\_priority = 0

Show SCHED\_FIFO of priority

max\_priority = 99

min\_priority = 1

Show SCHED\_RR of priority

max\_priority = 99

min\_priority = 1

Show priority of current thread

priority = 0

**\_\_scope**，表示线程间竞争CPU的范围，也就是说线程优先级的有效范围。POSIX的标准中定义了两个值：PTHREAD\_SCOPE\_SYSTEM和PTHREAD\_SCOPE\_PROCESS，前者表示与系统中所有线程一起竞争CPU时间，后者表示仅与同进程中的线程竞争CPU。目前LinuxThreads仅实现了PTHREAD\_SCOPE\_SYSTEM一值。

函数pthread\_attr\_setscope和pthread\_attr\_getscope分别用来设置和得到线程的作用域。

#include <pthread.h>

int pthread\_attr\_getscope( const pthread\_attr\_t \* attr, int \* scope );

int pthread\_attr\_setscope( pthread\_attr\_t\*, int scope );

参数：  
attr 线程属性变量  
scope 线程的作用域  
若成功返回0，若失败返回-1。

作用域控制线程是否在进程内或在系统级上竞争资源，可能的值是  
PTHREAD\_SCOPE\_PROCESS（进程内竞争资源）  
PTHREAD\_SCOPE\_SYSTEM （系统级竞争资源）。

为了设置这些属性，POSIX定义了一系列属性设置函数，包括pthread\_attr\_init()、pthread\_attr\_destroy()和与各个属性相关的pthread\_attr\_getXXX/pthread\_attr\_setXXX函数。

主要的函数如下：

1、pthread\_attr\_init  
功能： 对线程属性变量的初始化。  
头文件： <pthread.h>  
函数原型： int pthread\_attr\_init (pthread\_attr\_t\* attr);  
函数传入值：attr:线程属性。  
函数返回值：成功： 0  
失败： -1

2、pthread\_attr\_setscope  
功能： 设置线程 \_\_scope 属性。scope属性表示线程间竞争CPU的范围，也就是说线程优先级的有效范围。POSIX的标准中定义了两个值：PTHREAD\_SCOPE\_SYSTEM和PTHREAD\_SCOPE\_PROCESS，前者表示与系统中所有线程一起竞争CPU时间，后者表示仅与同进程中的线程竞争CPU。默认为PTHREAD\_SCOPE\_PROCESS。目前LinuxThreads仅实现了PTHREAD\_SCOPE\_SYSTEM一值。

头文件： <pthread.h>  
函数原型： int pthread\_attr\_setscope (pthread\_attr\_t\* attr, int scope);  
函数传入值：attr: 线程属性。  
scope:PTHREAD\_SCOPE\_SYSTEM，表示与系统中所有线程一起竞争CPU时间，  
PTHREAD\_SCOPE\_PROCESS，表示仅与同进程中的线程竞争CPU  
函数返回值得：同1。

3、pthread\_attr\_setdetachstate  
功能： 设置线程detachstate属性。该表示新线程是否与进程中其他线程脱离同步，如果设置为PTHREAD\_CREATE\_DETACHED则新线程不能用pthread\_join()来同步，且在退出时自行释放所占用的资源。缺省为PTHREAD\_CREATE\_JOINABLE状态。这个属性也可以在线程创建并运行以后用pthread\_detach()来设置，而一旦设置为PTHREAD\_CREATE\_DETACH状态（不论是创建时设置还是运行时设置）则不能再恢复到PTHREAD\_CREATE\_JOINABLE状态。

头文件： <phread.h>  
函数原型： int pthread\_attr\_setdetachstate (pthread\_attr\_t\* attr, int detachstate);  
函数传入值：attr:线程属性。  
detachstate:PTHREAD\_CREATE\_DETACHED，不能用pthread\_join()来同步，且在退出时自行释放所占用的资源  
PTHREAD\_CREATE\_JOINABLE，能用pthread\_join()来同步  
函数返回值得：同1。

4、pthread\_attr\_setschedparam  
功能： 设置线程schedparam属性，即调用的优先级。

头文件： <pthread.h>  
函数原型： int pthread\_attr\_setschedparam (pthread\_attr\_t\* attr, struct sched\_param\* param);  
函数传入值：attr：线程属性。  
param：线程优先级。一个struct sched\_param结构，目前仅有一个sched\_priority整型变量表示线程的运行优先级。这个参数仅当调度策略为实时（即SCHED\_RR或SCHED\_FIFO）时才有效，并可以在运行时通过pthread\_setschedparam()函数来改变，缺省为0  
函数返回值：同1。

5、pthread\_attr\_getschedparam  
功能： 得到线程优先级。  
头文件： <pthread.h>  
函数原型： int pthread\_attr\_getschedparam (pthread\_attr\_t\* attr, struct sched\_param\* param);

函数传入值：

attr：线程属性；  
param：线程优先级；

函数返回值：同1。

线程属性程序eg：

pthread\_t thd;void \*thread(void \*arg);

int main(){

pthread\_attr\_t attr={0};

pthread\_attr\_init(&attr);

struct sched\_param sched={99};

pthread\_attr\_setschedpolicy(&attr,SCHED\_RR);

pthread\_attr\_setschedparam(&attr,&sched);

if(pthread\_create(&thd,&attr,thread,NULL)!=0)

{

pthread\_attr\_destroy(&attr);

printf("create thread error\n");

return;

}}

1. 线程可以通过调用pthread\_self函数获得自身线程ID

pthread\_t pthread\_self(void)

7.

# 二．Linux中CPU亲和性（affinity）

**1、准备知识：**

**并行：**两件（多件）事情在同一时刻一起发生。

**并发：**两件（多件）事情在同一时刻只能有一个发生，由于CPU快速切换，从而给人的感觉是同时进行。

**单核多线程：**单核CPU上运行多线程， 同一时刻只有一个线程在跑，系统进行线程切换，系统给每个线程分配时间片来执行，看起来就像是同时在跑， 但实际上是每个线程跑一点点就换到其它线程继续跑。

**多核多线程：**每个核上各自运行线程，同一时刻可以有多个线程同时在跑。

**物理CPU：**机器上实际安装的CPU个数，比如说你的主板上安装了一块8核CPU，那么物理CPU个数就是1个，所以物理CPU个数就是主板上安装的CPU个数。

**逻辑CPU：**一般情况，我们认为一颗CPU可以有多个核，加上intel的超线程技术(HT), 可以在逻辑上再分一倍数量的CPU core出来。

**超线程技术(Hyper-Threading)：**就是利用特殊的硬件指令，把两个逻辑CPU模拟成两个物理CPU，实现多核多线程。我们常听到的双核四线程/四核八线程指的就是支持超线程技术的CPU。

逻辑CPU数量 = 物理CPU数量 x CPU cores x 2(如果支持并开启HT)  
比如：你的电脑安装了一块4核CPU，并且支持且开启了超线程（HT）技术，那么逻辑CPU数量 = 1 × 4 × 2 = 8

**2、CPU的亲和性（affinity）**

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。

**软亲和性（affinity）:** 就是进程要在指定的 CPU 上尽量长时间地运行而不被迁移到其他处理器，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。

**3、什么是 Linux 内核硬亲和性（affinity）？**

**硬亲和性（affinity）：**简单来说就是利用linux内核提供给用户的API，强行将进程或者线程绑定到某一个指定的cpu核运行。

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。

**4、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. 提高Cache命中率**

在多核运行的机器上，每个CPU都有自己的缓存，缓存着进程使用的信息，而进程可能会被OS调度到其他CPU上，如此一来CPU Cache命中率就低了。当绑定CPU后，程序就会一直在指定的cpu跑，不会由OS调度到其他CPU上，提高CPU Cache命中率。

**原因 3. 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。

**5、如何设置硬亲和性（affinity）？**

**（1）.用户态进程与CPU绑定**

**函数原型：**

int sched\_setaffinity(pid\_t pid, size\_t cpusetsize, cpu\_set\_t \*mask);

该函数设置进程为pid的这个进程,让它运行在mask所设定的CPU上.如果pid的值为0，则表示指定的是当前进程,使当前进程运行在mask所设定的那些CPU上。第二个参数cpusetsize是mask所指定的数的长度.通常设定为sizeof(cpu\_set\_t)。如果当前pid所指定的进程此时没有运行在mask所指定的任意一个CPU上，则该指定的进程会从其它CPU上迁移到mask的指定的一个CPU上运行。

int sched\_getaffinity(pid\_t pid, size\_t cpusetsize, cpu\_set\_t \*mask);

该函数获得pid所指示的进程的CPU位掩码，并将该掩码返回到mask所指向的结构中。即获得指定pid当前可以运行在哪些CPU上。同样，如果pid的值为0。也表示的是当前进程。

**宏定义：**

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

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);

将指定的 cpu 加入 CPU 集 set 中

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

将指定的 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）。

**相关函数：**

long sysconf(int name);

**参数name取值：**

\_SC\_NPROCESSORS\_CONF：查看cpu的个数

\_SC\_NPROCESSORS\_ONLN：查看正在使用的cpu个数

\_SC\_PAGESIZE：查看缓存内存页面的大小

\_SC\_PHYS\_PAGES：查看内存的总页数

\_SC\_AVPHYS\_PAGES：查看可以利用的总页数

\_SC\_LOGIN\_NAME\_MAX：查看最大登录名长度

\_SC\_HOST\_NAME\_MAX：查看最大主机名长度

\_SC\_OPEN\_MAX：每个进程运行时打开的文件数目

\_SC\_CLK\_TCK：查看每秒中跑过的运算速率

sysconf(\_SC\_PAGESIZE) \* sysconf(\_SC\_PHYS\_PAGES) ：计算内存大小

**伪代码：**

cpu\_set\_t mask;

/\* 初始化set集，将set设置为空\*/

CPU\_ZERO(&mask);

/\* 依次将0、1号cpu加入到集合\*/

CPU\_SET(0, &mask);

CPU\_SET(1, &mask);

/\*将当前进程绑定到cpu \*/

sched\_setaffinity(0, sizeof(mask), &mask);

**（2）.用户态线程与CPU绑定**

**函数原型**

int pthread\_setaffinity\_np(pthread\_t thread, size\_t cpusetsize,const cpu\_set\_t \*cpuset);

int pthread\_getaffinity\_np(pthread\_t thread, size\_t cpusetsize,cpu\_set\_t \*cpuset);

**相关函数：**

pthread\_t pthread\_self(void);

功能是获得线程自身的ID。

**伪代码：**

cpu\_set\_t mask;

/\* 初始化set集，将set设置为空\*/

CPU\_ZERO(&mask);

/\* 依次将0、1号cpu加入到集合\*/

CPU\_SET(0, &mask);

CPU\_SET(1, &mask);

/\*将当前线程程绑定到cpu \*/

pthread\_setaffinity\_np(pthread\_self(), sizeof(mask), &mask);