# Pthread:

## 一、基础

Linux多线程程序需要包含头文件

#include <pthread.h>

还需要链接libpthread.so这个库，在程序链接阶段应该有类似：

gcc program.o -o program -lpthread

Linux用户级线程主要解决上下文切换问题，调度算法与调度过程由用户决定，POSIX提供的用户级线程接口：

I.线程的建立结束

II.线程的互斥与同步

III.使用信号量控制线程

IV.线程基本属性配置

### A．基本线程操作：

1.pthread\_create() 创建线程开始运行相关线程函数，运行结束则线程退出

完整定义：

int pthread\_create(pthread\_t \* thread, const pthread\_attr\_t \* attr, void \*(\*start\_routine)(void \*), void \*arg);

函数参数：

（1）线程句柄 thread：当一个新的线程调用成功之后，就会通过这个参数将线程的句柄返回给调用者，以便对这个线程进行管理。

（2）入口函数 start\_routine()： 当你的程序调用了这个接口之后，就会产生一个线程，而这个线程的入口函数就是start\_routine()。如果线程创建成功，这个接口会返回0。

（3）入口函数参数 \*arg : start\_routine()函数有一个参数，这个参数就是pthread\_create的最后一个参数arg。这种设计可以在线程创建之前就帮它准备好一些专有数据，最典型的用法就是使用C++编程时的this指针。start\_routine()有一个返回值，这个返回值可以通过pthread\_join()接口获得。

（4）线程属性 attr： pthread\_create()接口的第二个参数用于设置线程的属性。这个参数是可选的，当不需要修改线程的默认属性时，给它传递NULL就行。具体线程有那些属性，我们后面再做介绍。

代码：

#include <stdio.h>

#include <pthread>

void \*thread(void \* arg){

printf("This is a Thread and arg = %d.\n", \*(int \*)arg);

\*(int \*)arg = 0;

}

int main(int argc, char \*argv[]){

pthread\_t th;

int ret;

int arg = 10;

int \*thread\_ret = NULL;

ret = pthread\_create(&th, NULL, thread, &arg);

if(ret != 0){

printf("Create thread error.\n");

return -1;

}

printf("This is the main process.\n");

pthread\_join(th, (void \*\*)&thread\_ret);

printf("thread\_ret = %d.\n", \*thread\_ret);

return 0;

}

保存代码为thread.c执行：

gcc thread.c -o thread -lpthread

./thread

不同环境下可能有不同结果，线程代码可能先于第24行代码被执行

代码分析：

-在第18行调用pthread\_create()接口创建了一个新的线程，这个线程的入口函数是thread()，并且给这个入口函数传递了一个参数，且参数值为10。

-这个新创建的线程要执行的任务非常简单，只是将显示“This is a thread and arg = 10”这个字符串，因为arg这个参数值已经定义好了，就是10。之后线程将arg参数的值修改为0，并将它作为线程的返回值返回给系统。

-pthread\_join()这个接口的第一个参数就是新创建线程的句柄了，而第二个参数就会去接受线程的返回值。pthread\_join()接口会阻塞主进程的执行，直到合并的线程执行结束。由于线程在结束之后会将0返回给系统，那么pthread\_join()获得的线程返回值自然也就是0。输出结果“thread\_ret = 0”也证实了这一点。

代码：

/\* thread.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define THREAD\_NUMBER 3 /\*线程数\*/

#define REPEAT\_NUMBER 5 /\*每个线程中的小任务数\*/

#define DELAY\_TIME\_LEVELS 10.0 /\*小任务之间的最大时间间隔\*/

//

void \*thrd\_func(void \*arg) {

/\* 线程函数例程 \*/

int thrd\_num = (int)arg;

int delay\_time = 0;

int count = 0;

printf("Thread %d is starting\n", thrd\_num);

for (count = 0; count < REPEAT\_NUMBER; count++) {

delay\_time = (int)(rand() \* DELAY\_TIME\_LEVELS/(RAND\_MAX)) + 1;

sleep(delay\_time);

printf("\tThread %d: job %d delay = %d\n",

thrd\_num, count, delay\_time);

}

printf("Thread %d finished\n", thrd\_num);

pthread\_exit(NULL);

}

int main(void) {

pthread\_t thread[THREAD\_NUMBER];

int no = 0, res;

void \* thrd\_ret;

srand(time(NULL));

for (no = 0; no < THREAD\_NUMBER; no++) {

/\* 创建多线程 \*/

res = pthread\_create(&thread[no], NULL, thrd\_func, (void\*)no);

if (res != 0) {

printf("Create thread %d failed\n", no);

exit(res);

}

}

printf("Create treads success\n Waiting for threads to finish...\n");

for (no = 0; no < THREAD\_NUMBER; no++) {

/\* 等待线程结束 \*/

res = pthread\_join(thread[no], &thrd\_ret);

if (!res) {

printf("Thread %d joined\n", no);

} else {

printf("Thread %d join failed\n", no);

}

}

return 0;

}

例程中循环3次建立3条线程，并且使用pthread\_join函数依次等待线程结束；

线程中使用rand()获得随机值随机休眠5次，随意会出现后执行的线程先执行完成；

运行结果：

$ gcc thread.c -lpthread

$ ./a.out

Create treads success

Waiting for threads to finish...

Thread 0 is starting

Thread 1 is starting

Thread 2 is starting

Thread 1: job 0 delay = 2

Thread 1: job 1 delay = 2

Thread 0: job 0 delay = 8

Thread 2: job 0 delay = 10

Thread 2: job 1 delay = 3

Thread 1: job 2 delay = 10

Thread 0: job 1 delay = 8

Thread 0: job 2 delay = 3

Thread 0: job 3 delay = 1

Thread 2: job 2 delay = 8

Thread 1: job 3 delay = 8

Thread 1: job 4 delay = 1

Thread 1 finished

Thread 2: job 3 delay = 6

Thread 0: job 4 delay = 7

Thread 0 finished

Thread 0 joined

Thread 1 joined

Thread 2: job 4 delay = 10

Thread 2 finished

Thread 2 joined

线程1先于线程0执行，但是pthread\_join的调用时间顺序，先等待线程0执行。由于线程1已经早结束，所以线程0被pthread\_join等到的时候，线程1已经结束，就在等待到线程1时，直接返回。

2.pthread\_eixt() 因为exit()是用来结束进程的，所以则需要使用特定结束线程的函数

3.pthread\_join() 挂起当前线程，用于阻塞式地等待线程结束，如果线程已结束则立即返回，0=成功

#### 线程的合并

创建线程之后，线程属于系统资源，占用内存空间，有创建要有回收，否则会有资源泄露问题（参照malloc之后不进行free产生内存泄漏问题）。线程的合并是一种主动回收线程资源的方案，当一个进程或线程调用了针对其他线程的pthread\_join()接口，就是线程合并了。此接口会阻塞调用进程或线程，直到被合并的线程结束为止。当被合并线程结束，pthread\_join()接口会回收这个线程的资源，并将这个线程的返回值返回给合并者。

#### 线程的分离

与线程合并相对应的另外一种线程资源回收机制是线程分离，调用接口pthread\_detach()。线程的分离是将线程的工作交由系统自动来完成，也即当被分离的线程结束之后，系统会自动回收它的资源。因为线程分离是启动系统的自动回收机制，那么程序也就无法获得被分离线程的返回值，使得pthread\_detach()接口只要有一个参数即可，即被分离线程的句柄。

线程的合并与分离均用于回收线程资源。

4.pthread\_cancel() 发送终止信号给thread线程，成功返回0，但是成功并不意味着thread会终止

5.pthread\_testcancel() 在不包含取消点，但是又需要取消点的地方创建一个取消点，以便在一个没有包含取消点的执行代码线程中响应取消请求.

6.pthread\_setcancelstate() 设置本线程对Cancel信号的反应

7.pthread\_setcanceltype() 设置取消状态 继续运行至下一个取消点再退出或者是立即执行取消动作

8.pthread\_setcancel() 设置取消状态

### B．互斥与同步机制基本函数：

（接线程属性及相关配置中线程本地存储）虽然线程本地存储可以避免线程访问共享数据，但是线程之间的绝大多数数据始终还是共享的。在涉及对共享数据进行读写操作时，就必须使用

1.pthread\_mutex\_init() 互斥锁的初始化

/\*thread\_mutex.c\*/

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define THREAD\_NUMBER 3 /\* 线程数 \*/

#define REPEAT\_NUMBER 3 /\* 每个线程的小任务数 \*/

#define DELAY\_TIME\_LEVELS 10.0 /\*小任务之间的最大时间间隔\*/

pthread\_mutex\_t mutex;

void \*thrd\_func(void \*arg) {

int thrd\_num = (int)arg;

int delay\_time = 0, count = 0;

int res;

/\* 互斥锁上锁 \*/

res = pthread\_mutex\_lock(&mutex);

if (res) {

printf("Thread %d lock failed\n", thrd\_num);

pthread\_exit(NULL);

}

printf("Thread %d is starting\n", thrd\_num);

for (count = 0; count < REPEAT\_NUMBER; count++) {

delay\_time = (int)(rand() \* DELAY\_TIME\_LEVELS/(RAND\_MAX)) + 1;

sleep(delay\_time);

printf("\tThread %d: job %d delay = %d\n",

thrd\_num, count, delay\_time);

}

printf("Thread %d finished\n", thrd\_num);

pthread\_exit(NULL);

}

int main(void) {

pthread\_t thread[THREAD\_NUMBER];

int no = 0, res;

void \* thrd\_ret;

srand(time(NULL));

/\* 互斥锁初始化 \*/

pthread\_mutex\_init(&mutex, NULL);

for (no = 0; no < THREAD\_NUMBER; no++) {

res = pthread\_create(&thread[no], NULL, thrd\_func, (void\*)no);

if (res != 0) {

printf("Create thread %d failed\n", no);

exit(res);

}

}

printf("Create treads success\n Waiting for threads to finish...\n");

for (no = 0; no < THREAD\_NUMBER; no++) {

res = pthread\_join(thread[no], &thrd\_ret);

if (!res) {

printf("Thread %d joined\n", no);

} else {

printf("Thread %d join failed\n", no);

}

}

/\*\*\*\*互斥锁解锁\*\*\*/

pthread\_mutex\_unlock(&mutex);

pthread\_mutex\_destroy(&mutex);

return 0;

}

添加同步锁pthread\_mutex\_t;

在线程中加入，于是程序在执行线程程序时；调用pthread\_mutex\_lock上锁，发现上锁时候进入等待，等待锁再次释放后重新上锁；

所以线程程序加载到队列中等待，等待成功上锁后继续执行程序代码

运行结果：

$gcc thread\_mutex.c -lpthread

$ ./a.out

Create treads success

Waiting for threads to finish...

Thread 0 is starting

Thread 0: job 0 delay = 9

Thread 0: job 1 delay = 4

Thread 0: job 2 delay = 7

Thread 0 finished

Thread 0 joined

Thread 1 is starting

Thread 1: job 0 delay = 6

Thread 1: job 1 delay = 4

Thread 1: job 2 delay = 7

Thread 1 finished

Thread 1 joined

Thread 2 is starting

Thread 2: job 0 delay = 3

Thread 2: job 1 delay = 1

Thread 2: job 2 delay = 6

Thread 2 finished

Thread 2 joined

线程程序被加载到队列中而不能马上执行，需要等到能够成功上锁；

上锁后，继续执行线程程序，上锁执行；

使用场景：

如：当用户登录后获取秘钥才能够继续获取该用户的基本信息时；需等待登录线程结束后才能继续执行获取用户信息的线程时，需要调用两条线程，假如是：threadLogin()与threadGetinfo()；则有两种方法实现。

1. 使用互斥锁一次性调用完threadLogin(),threadGetinfo()
2. 不适用互斥锁，直接在threadLogin()登录验证中成功之后调用threadGetinfo()，相比之下，a方法更好，在逻辑关系方面，代码可读性与可扩展性。

2.pthread\_mutex\_lock() 锁定互斥锁，如果尝试锁定已经被上锁的互斥锁则阻塞至可用为止

3.pthread\_mutex\_trylock() 非阻塞的锁定互斥锁

4.pthread\_mutex\_unlock() 释放互斥锁

5.pthread\_mutex\_destory() 互斥锁销毁函数

### C．信号量线程控制(默认无名信号量):

sem\_init(sem) 初始化一个定位在sem的匿名信号量

sem\_wait() 把信号量减1操作，如果信号量的当前值为0则进入阻塞，为原子操作

sem\_trywait() 如果信号量的当前值为0则返回错误而不是阻塞调用(errno=EAGAIN),其实是sem\_wait()的非阻塞版本

sem\_post() 给信号量的值加1，它是一个“原子操作”，即同时对同一个信号量做加1,操作的两个线程是不会冲突的

sem\_getvalue(sval) 把sem指向的信号量当前值放置在sval指向的整数上

sem\_destory(sem) 销毁由sem指向的匿名信号量

### D．线程属性配置相关函数：

1.pthread\_attr\_init()初始化配置一个线程对象的属性,需要用pthread\_attr\_destroy函数去除已有属性

#### 线程的属性：

线程的属性由一个线程属性对象来描述。线程属性对象由pthread\_attr\_init()接口初始化，并由pthread\_attr\_destory()来销毁

完整定义：

int pthread\_attr\_init(pthread\_attr\_t \*attr);

int pthread\_attr\_destory(pthread\_attr\_t \*attr);

Linux下线程的属性一般包括：绑定属性、分离属性、调度属性、堆栈大小属性和满占警戒区大小属性。

#### 绑定属性：

说到这个绑定属性，就不得不提起另外一个概念：轻进程（Light Weight Process，简称LWP）。

轻进程和Linux系统的内核线程拥有相同的概念，属于内核的调度实体。一个轻进程可以控制一个或多个线程。

在计算机操作系统中,轻量级进程（LWP）是一种实现多任务的方法。与普通进程相比，LWP与其他进程共享所有（或大部分）它的逻辑地址空间和系统资源；与线程相比，LWP有它自己的进程标识符，并和其他进程有着父子关系；这是和类Unix操作系统的系统调用vfork()生成的进程一样的。另外，线程既可由应用程序管理，又可由内核管理，而LWP只能由内核管理并像普通进程一样被调度。Linux内核是支持LWP的典型例子。

默认情况下，对于一个拥有n个线程的程序，启动多少轻进程，由哪些轻进程来控制哪些线程由操作系统来控制，这种状态被称为非绑定的。那么绑定的含义就很好理解了，只要指定了某个线程“绑”在某个轻进程上，就可以称之为绑定的了。

被绑定的线程具有较高的相应速度，因为操作系统的调度主体是轻进程，绑定线程可以保证在需要的时候它总有一个轻进程可用。绑定属性就是干这个用的。

设置绑定属性的接口是pthread\_attr\_setscope()，

完整定义：

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

它有两个参数，第一个就是线程属性对象的指针，第二个就是绑定类型，拥有两个取值：PTHREAD\_SCOPE\_SYSTEM（绑定的）和PTHREAD\_SCOPE\_PROCESS（非绑定的）。代码2演示了这个属性的使用。

代码2 : 设置线程绑定属性

#include <stdio.h>

#include <pthread.h>

//

int main(int argc, char \* argv[]){

pthread\_attr\_t attr;

pthread\_t th;

//

pthread\_attr\_init(&attr);

pthread\_attr\_setscope(&attr, PTHREAD\_SCOPE\_SYSTEM);

pthread\_create(&th, &attr, thread, NULL);

//

}

不知道你是否在这里发现了本文的矛盾之处。就是这个绑定属性跟我们之前说的NPTL有矛盾之处。

在介绍NPTL的时候就说过业界有一种m:n的线程方案，就跟这个绑定属性有关。但是笔者还说过NPTL因为Linux的“蠢”没有采取这种方案，而是采用了“1:1”的方案。这也就是说，Linux的线程永远都是绑定。

对，Linux的线程永远都是绑定的，所以PTHREAD\_SCOPE\_PROCESS在Linux中不管用，而且会返回ENOTSUP错误。

既然Linux并不支持线程的非绑定，为什么还要提供这个接口呢？答案就是兼容！因为Linux的NTPL是号称POSIX标准兼容的，而绑定属性正是POSIX标准所要求的，所以提供了这个接口。如果读者们只是在Linux下编写多线程程序，可以完全忽略这个属性。如果哪天你遇到了支持这种特性的系统，别忘了我曾经跟你说起过这玩意儿：）

#### 分离属性:

前面说过线程能够被合并和分离，分离属性就是让线程在创建之前就决定它应该是分离的。如果设置了这个属性，就没有必要调用pthread\_join()或pthread\_detach()来回收线程资源了。

设置分离属性的接口是pthread\_attr\_setdetachstate()，

完整定义：

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

它的第二个参数有两个取值：PTHREAD\_CREATE\_DETACHED（分离的）和PTHREAD\_CREATE\_JOINABLE（可合并的，也是默认属性）。代码3演示了这个属性的使用。

代码3 : 设置线程分离属性

#include <stdio.h>

#include <pthread.h>

//

int main(int argc, char \* argv[]){

pthread\_attr\_t attr;

pthread\_t th;

//

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstat(&attr, PTHREAD\_CREATE\_DETACHED);

pthread\_create(&th, &attr, thread, NULL);

//

}

#### 调度属性:

线程的调度属性有三个，分别是：算法、优先级和继承权。

算法:

Linux提供的线程调度算法有三个：轮询、先进先出和其它。

其中轮询和先进先出调度算法是POSIX标准所规定，而其他则代表采用Linux自己认为更合适的调度算法，所以默认的调度算法也就是其它了。

轮询和先进先出调度算法都属于**实时调度算法。**

轮询指的是时间片轮转，当线程的时间片用完，系统将重新分配时间片，并将它放置在就绪队列尾部，这样可以保证具有相同优先级的轮询任务获得公平的CPU占用时间；

先进先出就是先到先服务，一旦线程占用了CPU则一直运行，直到有更高优先级的线程出现或自己放弃。

设置线程调度算法的接口是pthread\_attr\_setschedpolicy()，

完整定义：

pthread\_attr\_setschedpolicy(pthread\_attr\_t \*attr, int policy);

它的第二个参数有三个取值：SCHED\_RR（轮询）、SCHED\_FIFO（先进先出）和SCHED\_OTHER（其它）。

#### 优先级:

Linux的线程优先级与进程的优先级不一样，进程优先级我们后面再说。

Linux的线程优先级是从1到99的数值，数值越大代表优先级越高。

而且要注意的是，只有采用SHCED\_RR或SCHED\_FIFO调度算法时，优先级才有效。对于采用SCHED\_OTHER调度算法的线程，其优先级恒为0。

设置线程优先级的接口是pthread\_attr\_setschedparam()，

完整定义：

struct sched\_param{

int sched\_priority;

}

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

sched\_param结构体的sched\_priority字段就是线程的优先级了。

此外，即便采用SCHED\_RR或SCHED\_FIFO调度算法，线程优先级也不是随便就能设置的。首先，进程必须是以root账号运行的；其次，还需要放弃线程的继承权。什么是继承权呢？

继承权:

继承权就是当创建新的线程时，新线程要继承父线程（创建者线程）的调度属性。如果不希望新线程继承父线程的调度属性，就要放弃继承权。

设置线程继承权的接口是pthread\_attr\_setinheritsched()，它的完整定义是：int pthread\_attr\_setinheritsched(pthread\_attr\_t \*attr, int inheritsched);

它的第二个参数有两个取值：PTHREAD\_INHERIT\_SCHED（拥有继承权）和PTHREAD\_EXPLICIT\_SCHED（放弃继承权）。新线程在默认情况下是拥有继承权。

代码4能够演示不同调度算法和不同优先级下各线程的行为，同时也展示如何修改线程的调度属性。

代码4：设置线程调度属性:

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <pthread.h>

#define THREAD\_COUNT 12

void show\_thread\_policy(int threadno)

{

int policy;

struct sched\_param param;

pthread\_getschedparam(pthread\_self(), &policy, param);

switch(policy){

case SCHED\_OTHER:

printf("SCHED\_OTHER %d\n", threadno);

break;

case SCHED\_RR:

printf("SCHDE\_RR %d\n", threadno);

break;

case SCHED\_FIFO:

printf("SCHED\_FIFO %d\n", threadno);

break;

default:

printf("UNKNOWN\n");

}

}

void\* thread(void \*arg)

{

int i, j;

long threadno = (long)arg;

printf("thread %d start\n", threadno);

sleep(1);

show\_thread\_policy(threadno);

for( i = 0; i < 10; ++i ) {

for( j = 0; j < 100000000; ++j ){}

printf("thread %d\n", threadno);

}

printf("thread %d exit\n", threadno);

return NULL;

}

int main( int argc, char \*argv[] )

{

long i;

pthread\_attr\_t attr[THREAD\_COUNT];

pthread\_t pth[THREAD\_COUNT];

struct sched\_param param;

for( i = 0; i < THREAD\_COUNT; ++i )

pthread\_attr\_init( &attr[i] );

for( i = 0; i < THREAD\_COUNT / 2; ++i ) {

param.sched\_priority = 10;

pthread\_attr\_setschedpolicy( &attr[i], SCHED\_FIFO );

pthread\_attr\_setschedparam( &attr[i], param );

pthread\_attr\_setinheritsched(&attr[i], PTHREAD\_EXPLICIT\_SCHED);

}

for( i = THREAD\_COUNT / 2; i < THREAD\_COUNT; ++i ) {

param.sched\_priority = 20;

pthread\_attr\_setschedpolicy( &attr[i], SCHED\_FIFO );

pthread\_attr\_setschedparam( &attr[i], param );

pthread\_attr\_setinheritsched(&attr[i], PTHREAD\_EXPLICIT\_SCHED );

}

for( i = 0; i < THREAD\_COUNT; ++i )

pthread\_create( &pth[i], &attr[i], thread, (void\*)i );

for( i = 0; i < THREAD\_COUNT; ++i )

pthread\_join( pth[i], NULL );

for( i = 0; i < THREAD\_COUNT; ++i )

pthread\_attr\_destroy( &attr[i] );

return 0;

}

这段代码中含有一些没有介绍过的接口，读者们可以使用Linux的联机帮助来查看它们的具体用法和作用。

#### 堆栈大小属性:

从前面的这些例子中可以了解到，线程的主函数与程序的主函数main()有一个很相似的特性，那就是可以拥有局部变量。虽然同一个进程的线程之间是共享内存空间的，但是它的局部变量确并不共享。原因就是局部变量存储在堆栈中，而不同的线程拥有不同的堆栈。Linux系统为每个线程默认分配了8MB的堆栈空间，如果觉得这个空间不够用，可以通过修改线程的堆栈大小属性进行扩容。

修改线程堆栈大小属性的接口是pthread\_attr\_setstacksize()，

完整定义：

int pthread\_attr\_setstacksize(pthread\_attr\_t \*attr, size\_t stacksize);

它的第二个参数就是堆栈大小了，以字节为单位。需要注意的是，线程堆栈不能小于16KB，而且尽量按4KB(32位系统)或2MB（64位系统）的整数倍分配，也就是内存页面大小的整数倍。此外，修改线程堆栈大小是有风险的，如果你不清楚你在做什么，最好别动它（其实我很后悔把这么危险的东西告诉了你:）。

#### 满栈警戒区属性:

既然线程是有堆栈的，而且还有大小限制，那么就一定会出现将堆栈用满的情况。线程的堆栈用满是非常危险的事情，因为这可能会导致对内核空间的破坏，一旦被有心人士所利用，后果也不堪设想。为了防治这类事情的发生，Linux为线程堆栈设置了一个满栈警戒区。这个区域一般就是一个页面，属于线程堆栈的一个扩展区域。一旦有代码访问了这个区域，就会发出SIGSEGV信号进行通知。

虽然满栈警戒区可以起到安全作用，但是也有弊病，就是会白白浪费掉内存空间，对于内存紧张的系统会使系统变得很慢。所有就有了关闭这个警戒区的需求。同时，如果我们修改了线程堆栈的大小，那么系统会认为我们会自己管理堆栈，也会将警戒区取消掉，如果有需要就要开启它。

修改满栈警戒区属性的接口是pthread\_attr\_setguardsize()，

完整定义：

int pthread\_attr\_setguardsize(pthread\_attr\_t \*attr, size\_t guardsize);

它的第二个参数就是警戒区大小了，以字节为单位。与设置线程堆栈大小属性相仿，应该尽量按照4KB或2MB的整数倍来分配。当设置警戒区大小为0时，就关闭了这个警戒区。

虽然栈满警戒区需要浪费掉一点内存，但是能够极大的提高安全性，所以这点损失是值得的。而且一旦修改了线程堆栈的大小，一定要记得同时设置这个警戒区。

#### 线程本地存储：

内线程之间可以共享内存地址空间，线程之间的数据交换可以非常快捷，这是线程最显著的优点。但是多个线程访问共享数据，需要昂贵的同步开销，也容易造成与同步相关的BUG，更麻烦的是有些数据根本就不希望被共享，这又是缺点。

C程序库中的errno是个最典型的一个例子。errno是一个全局变量，会保存最后一个系统调用的错误代码。在单线程环境并不会出现什么问题。但是在多线程环境，由于所有线程都会有可能修改errno，这就很难确定errno代表的到底是哪个系统调用的错误代码了。这就是有名的“非线程安全（Non Thread-Safe）”的。

此外，从现代技术角度看，在很多时候使用多线程的目的并不是为了对共享数据进行并行处理（在Linux下有更好的方案，后面会介绍）。更多是由于多核心CPU技术的引入，为了充分利用CPU资源而进行并行运算（不互相干扰）。换句话说，大多数情况下每个线程只会关心自己的数据而不需要与别人同步。

为了解决这些问题，可以有很多种方案。比如使用不同名称的全局变量。但是像errno这种名称已经固定了的全局变量就没办法了。在前面的内容中提到在线程堆栈中分配局部变量是不在线程间共享的。但是它有一个弊病，就是线程内部的其它函数很难访问到。

目前解决这个问题的简便易行的方案是线程本地存储，即Thread Local Storage，简称TLS。利用TLS，errno所反映的就是本线程内最后一个系统调用的错误代码了，也就是线程安全的了。

Linux提供了对TLS的完整支持，通过下面这些接口来实现：

int pthread\_key\_create(pthread\_key\_t \*key, void (\*destructor)(void\*));

int pthread\_key\_delete(pthread\_key\_t key);

void\* pthread\_getspecific(pthread\_key\_t key);

int pthread\_setspecific(pthread\_key\_t key, const void \*value);

-pthread\_key\_create()接口用于创建一个线程本地存储区。

- key : 第一个参数用来返回这个存储区的句柄，需要使用一个全局变量保存，以便所有线程都能访问到。

- \*destructor : 第二个参数是线程本地数据的回收函数指针，如果希望自己控制线程本地数据的生命周期，这个参数可以传递NULL。

- pthread\_key\_delete()接口用于回收线程本地存储区。其唯一的参数就要回收的存储区的句柄。

- pthread\_getspecific()和pthread\_setspecific()这个两个接口分别用于获取和设置线程本地存储区的数据。这两个接口在不同的线程下会有不同的结果不同（相同的线程下就会有相同的结果），这也就是线程本地存储的关键所在。

代码5展示了如何在Linux使用线程本地存储，注意执行结果，分析本地线程存储的一些特性，以及内存回收的时机。

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define THREAD\_COUNT 10

pthread\_key\_t g\_key;

typedef struct thread\_data{

int thread\_no;

} thread\_data\_t;

void show\_thread\_data()

{

thread\_data\_t \*data = pthread\_getspecific(g\_key);

printf("Thread %d \n", data->thread\_no);

}

void\* thread(void \*arg)

{

thread\_data\_t \*data = (thread\_data\_t \*)arg;

printf("Start thread %d\n", data->thread\_no);

pthread\_setspecific(g\_key, data);

show\_thread\_data();

printf("Thread %d exit\n", data->thread\_no);

}

void free\_thread\_data(void \*arg)

{

thread\_data\_t \*data = (thread\_data\_t\*)arg;

printf("Free thread %d data\n", data->thread\_no);

free(data);

}

int main(int argc, char \*argv[])

{

int i;

pthread\_t pth[THREAD\_COUNT];

thread\_data\_t \*data = NULL;

pthread\_key\_create(&g\_key, free\_thread\_data);

for(i = 0; i < THREAD\_COUNT; ++i) {

data = malloc(sizeof(thread\_data\_t));

data->thread\_no = i;

pthread\_create(&pth[i], NULL, thread, data);

}

for(i = 0; i < THREAD\_COUNT; ++i)

pthread\_join(pth[i], NULL);

pthread\_key\_delete(g\_key);

return 0;

}

3.pthread\_attr\_setscope() 设置线程属性

4.pthread\_attr\_setschedparam() 设置线程优先级

5.pthread\_attr\_getschedparam() 获取线程优先级