**1. 线程属性：**

使用pthread\_attr\_t类型表示，我们需要对此结构体进行初始化，

初始化后使用，使用后还要进行去除初始化！

pthread\_attr\_init:初始化

pthread\_attr\_destory:去除初始化

**#include <pthread.h>**

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

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

**若成功返回0，若失败返回-1。**

pthread\_attr\_init之后，pthread\_t结构所包含的内容就是操作系统实现

支持的线程所有属性的默认值。

如果pthread\_attr\_init实现时为属性对象分配了动态内存空间，

pthread\_attr\_destroy还会用无效的值初始化属性对象，因此如果经

pthread\_attr\_destroy去除初始化之后的pthread\_attr\_t结构被

pthread\_create函数调用，将会导致其返回错误。

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;

下面主要讨论此结构体！！！

**2. 分离状态------用到**

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！这样更

方便！

#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**

**3. 线程的继承性---偶尔用到**

函数pthread\_attr\_setinheritsched和pthread\_attr\_getinheritsched分别用来设

置和得到线程的继承性!

#include <pthread.h>

int pthread\_attr\_getinheritsched(const pthread\_attr\_t \*attr,int \*inheritsched);

int pthread\_attr\_setinheritsched(pthread\_attr\_t \*attr,int inheritsched);

参数：

attr 线程属性变量

inheritsched 线程的继承性

若成功返回0，若失败返回-1。

请注意：

继承性决定调度的参数是从创建的进程中继承还是使用在schedpolicy和schedparam属性中显式设置的调度信息。

线程没有默认的继承值设置，所以如果关心线程的调度策略和参数，

只能手动设置！

**可设置参数：**

**PTHREAD\_INHERIT\_SCHED：新的线程继承创建线程的策略和参数！**

**PTHREAD\_EXPLICIT\_SCHED：新的线程继承策略和参数来自于schedpolicy和schedparam属性中显式设置的调度信息！**

**4. 调度策略------用到**

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--------一般

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。

**5. 调度参数-----用到**

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

使用：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！

**6. 线程的作用域------用到**

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）。

**7. 线程堆栈的大小------用到**

函数pthread\_attr\_setstackaddr和pthread\_attr\_getstackaddr分别用来设置和得

到线程堆栈的位置。

int pthread\_attr\_getstacksize(const pthread\_attr\_t \*,size\_t \* stacksize);

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

参数：

attr 线程属性变量

stacksize 堆栈大小

若成功返回0，若失败返回-1。

**8. 线程堆栈的地址**

#include <pthread.h>

int pthread\_attr\_getstackaddr(const pthread\_attr\_t \*attr,void \*\*stackaddf);

int pthread\_attr\_setstackaddr(pthread\_attr\_t \*attr,void \*stackaddr);

参数：

attr 线程属性变量

stackaddr 堆栈地址

若成功返回0，若失败返回-1。

注意：pthread\_attr\_getstackaddr已经过期，现在使用的是pthread\_attr\_getstack

**9. 警戒缓冲区**

函数pthread\_attr\_getguardsize和pthread\_attr\_setguardsize分别用来设置和得

到线程栈末尾的警戒缓冲区大小。

#include <pthread.h>

int pthread\_attr\_getguardsize(const pthread\_attr\_t \*restrict attr,size\_t \*restrict

guardsize);

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

若成功返回0，若失败返回-1。

值得注意：

线程属性guardsize控制着线程栈末尾之后以避免栈溢出的扩展内存

大小。这个属性默认设置为PAGESIZE个字节。可以把guardsize线

程属性设为0，从而不允许属性的这种特征行为发生：在这种情况

下不会提供警戒缓存区。同样地，如果对线程属性stackaddr作了

修改，系统就会认为我们会自己管理栈，并使警戒栈缓冲区机制无

效，等同于把guardsize线程属性设为0。

**例子1：**

int comm\_thread\_create(pthread\_t \*thread, pthread\_attr\_t \*pAttr, thread\_func func, void \*param)

{

pthread\_attr\_t thread\_attr; //定义线程属性结构体变量

pthread\_attr\_init(&thread\_attr); //对线程属性变量进行初始化

/\*属性不为空时\*/

if(pAttr != NULL)

{

struct sched\_param param; //定义参数结构体变量

pthread\_attr\_setscope(&thread\_attr, PTHREAD\_SCOPE\_SYSTEM); //与系统中所有线程一起竞争cpu时间

pthread\_attr\_setschedpolicy(&thread\_attr, SCHED\_FIFO);

param.sched\_priority = 99; //默认为0

pthread\_attr\_setschedparam(&thread\_attr, &param);

}

/\* 设置线程为分离模式 ,这种模式下不能用pthread\_join来同步\*/

pthread\_attr\_setdetachstate(&thread\_attr, PTHREAD\_CREATE\_DETACHED); //退出时自行释放所占资源

/\* 设置线程栈大小1MB \*/

pthread\_attr\_setstacksize(&thread\_attr, 1\*1024\*1024);

/\*线程创建函数:线程ID,属性, 线程函数,线程参数\*/

int ret = pthread\_create(thread, &thread\_attr, func, param);

if(ret != 0)

{

/\*销毁设置好的线程属性\*/

pthread\_attr\_destroy(&thread\_attr);

}

return ret;

}