pthread_attr_t是控制线程属性的结构：

```
typedef struct __pthread_attr_s
{
```

int __detachstate; 设置线程是否和其他线程同步(其他线程能否调用pthread_join())，也可以在新线程运行中调用pthread_detach()完成。有两个值，PTHREAD_CREATE_DETACH和PTHREAD_CREATE_JOINABLE,默认值是后者,后者情况下线程的资源在退出后自行释放。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。

int __schedpolicy; 线程的调度策略，可以用pthread_setschedparam设置，有效值为SCHED_OTHER（正常、非实时）、SCHED_RR（实时、轮转法）和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。

struct __sched_param __schedparam; 调度参数，目前仅有一个sched_priority整型变量表示线程的运行优先级，表示线程的优先级，只在调度策略为SCHED_RR或SCHED_FIFO有效，并可以在运行时通过pthread_setschedparam()函数来改变，缺省为0

int __inheritsched; 有两种值可供选择：PTHREAD_EXPLICIT_SCHED和PTHREAD_INHERIT_SCHED，前者表示新线程使用显式指定调度策略和调度参数(即attr中的值)，而后者表示继承调用者线程的值。缺省为PTHREAD_EXPLICIT_SCHED。

int __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。

size_t __guardsize;

int __stackaddr_set;

void *__stackaddr;

size_t __stacksize;表示堆栈的大小。

}pthread_attr_t;

设置pthread_attr_t的相关函数:

初始化:

pthread_attr_init(&attr)

初始化的值为:

__scope = PTHREAD_SCOPE_PROCESS

__tetachstate = PTHREAD_CREATE_JOINABL

__stackaddr = NULL

__stacksize = 1M

__sched_param.priority = 0　　（使用创建线程的优先级）

__inheritsched = PTHREAD_INHERIT_SCHED

__schedpolicy = SCHED_OTHER

反初始化：

pthread_attr_destroy(&attr)

设置关联标志：

int　　pthread_attr_setdetachstate(pthread_attr_t *tattr, int detachstate);

detachstate = PTHREAD_CREATE_JOINABLE/PTHREAD_CREATE_DETACHED

设置PTHREAD_CREATE_DETACHED表示线程在退出后资源自动释放，不需要调用pthread_join

查询关联标志：

int pthread_attr_getdetachstate(const pthread_attr_t *tattr, int *detachstate;)

设置cpu竞争模式:

```
int pthread_attr_setscope(pthread_attr_t *tattr,int scope);
scope = PTHREAD_SCOPE_SYSTEM/PTHREAD_SCOPE_PROCESS
```

查询cpu竞争模式:

```
int pthread_attr_getscope(pthread_attr_t *tattr, int scope);
```

设置调度策略:

```
int pthread_attr_setschedpolicy(pthread_attr_t *tattr, int policy);
policy = SCHED_OTHER/SCHED_RR/SCHED_FIFO
```

查询调度策略:

```
int pthread_attr_getschedpolicy(pthread_attr_t *tattr, int policy);
```

设置继承模式:

```
int pthread_attr_setinheritsched(pthread_attr_t *tattr, int inherit);
```

inherit =
PTHREAD_INHERIT_SCHED/PTHREAD_EXPLICIT_SCHED

查询继承模式：

int pthread_attr_getinheritsched(pthread_attr_t *tattr, int inherit);

设置优先级：

int pthread_attr_setschedparam(pthread_attr_t *tattr, const struct sched_param *param);

查询优先级：

int pthread_attr_getschedparam(pthread_attr_t *tattr, const struct sched_param *param);

设置堆栈大小：

int pthread_attr_setstacksize(pthread_attr_t *tattr, int size);默认情况下线程保留1M的，而且会在堆栈的顶增加一个空闲的内存页，当访问该内存页的时候就会触发SIGSEGV信号，如果开发者设置了stack size那么就需要用户制定这个多余的内存页并且通过mprotect函数设置保护标志，而且它必须设置PTHREAD_CREATE_JOINABLE关联模式，

因为只有其他线程调用pthread_join后分配的资源才会被释放，线程的堆栈的分配必须大于一个最小值PTHREAD_STACK_MIN()。当分配内存的时候会设置MAP_NORESERVE标志(mmap)，这个标志表示不预留交换空间，当对该内存进行写的时候，如果系统不能分配到交换空间，那么就会触发SIGSEGV信号，如果可以分配到交换空间，那么就会把private page复制到交换空间。如果mmap没有指定MAP_NORESERVE，在分配空间的时候就会保留和映射区域相同大小的交换空间(这个其实就是资源的滞后分配原则)

查询堆栈大小：

```
int pthread_attr_getstacksize(pthread_attr_t *tattr,
int size);
```

设置堆栈地址：

int pthread_attr_setstackaddr(pthread_attr_t *tattr,void **stackaddr);如果线程地址为NULL，那么pthread分配指定的内存(1M)或者是指定的堆栈大小，如果设定了堆栈的地址那么内存的分配必须由开发者设定，例如：

```
stackbase = (void *) malloc(size);
ret = pthread_attr_setstacksize(&tattr, size);
```

```
ret = pthread_attr_setstackaddr(&tattr, stackbase);
ret = pthread_create(&tid, &tattr, func, arg);
```

查询堆栈地址：

```
int          pthread_attr_getstackaddr(pthread_attr_t
*tattr,void **stackaddr);
```

等待线程终止：

int pthread_join(thread_t tid, void **status);等待线程结束,这个函数会阻塞调用线程，如果多个线程同时等待一个线程，只有一个线程会成功返回，其他线程将会返回错误值ESRCH(无效的线程, 等待的线程).其他错误值包括：EDEADLK：自己等待自己; EINVAL: tid无效。

int pthread_detach(thread_t tid);将线程和其他线程脱离同步，别的线程不能对它调用pthread_join()，而且它的资源也是在退出时自行释放。

创建TSD key：

int pthread_key_create(pthread_key_t *key, void (*destructor) (void *));通常在创建线程前创建，然后在新创建线程中使用，void(*destructor)(void *)这是线程退出时如果TSD变量不是NULL，就会调用destructor。

pthread_key_create通常与pthread_once结合使用，以保证创建TSD只执行一次，如下：

```
static pthread_once_t key_only_one = PTHREAD_ONCE_INIT;
pthread_once(&key_only_one, key_create_function);
void key_create_function(void)
{
   pthread_key_create(&key_obj, free_key);
}

void free_key(void *arg)
{
   free(arg);
}
```

删除TSD key：

```
int pthread_key_delete(pthread_key_t *key);
```

设定TSD key对应的值：

```
int pthread_setspecific(pthread_key_t key, const void *value);
```