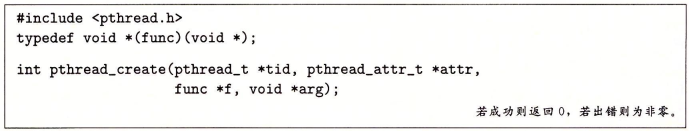


创建时预先派生指定数量的线程，然后去任务队列取添加进来的任务进行处理。

1、POSIX线程相关函数

**创建线程**



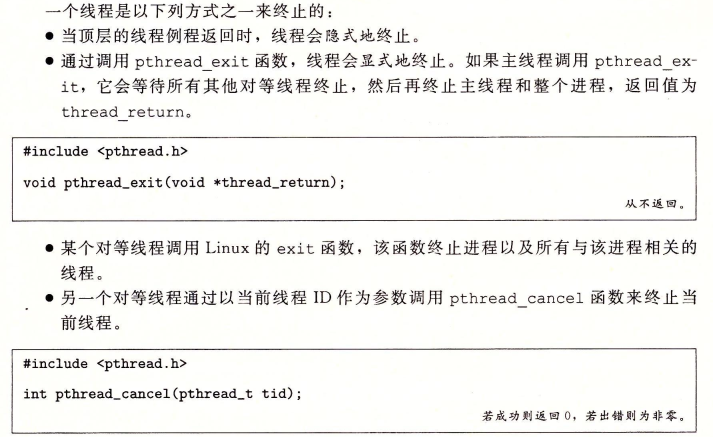
tid返回线程ID

attr修改线程属性

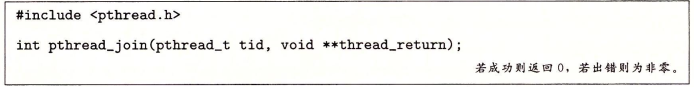
f在新线程中运行的函数

arg是输入变量

**终止线程**

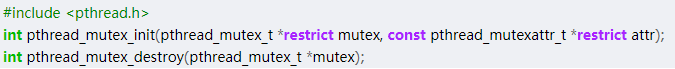


**回收线程资源**

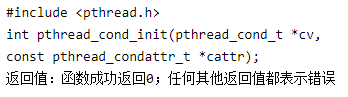


pthread\_join会阻塞，直到tid终止，将线程返回的通用（void \*）指针赋值为pthread\_return指向的位置，然后回收已终止线程占用的所有内存资源。

**互斥锁初始化**



**条件变量初始化**



以上两个初始化函数在第二个参数为NULL的时候使用默认属性。成功返回0.

2、数据结构

enum threadpool\_error\_t 枚举线程池错误

enum threadpool\_destroy\_flags\_t

enum threadpool\_shutdown\_t 枚举线程池关闭类型

struct threadpool\_task\_t 线程池任务结构

struct threadpool\_t 线程池

struct threadpool\_t {  
 pthread\_mutex\_t lock; //互斥锁  
 pthread\_cond\_t notify; //条件变量  
 pthread\_t \*threads; //线程池数组首指针  
 threadpool\_task\_t \*queue; //任务队列数组首指针  
 int thread\_count; //线程数量  
 int queue\_size; //任务数量  
 int head; //当前任务队列的头部  
 int tail; //当前任务队列的尾部  
 int count; //待运行的任务  
 int shutdown; //线程池状态是否关闭  
 int started; //正在运行的线程数  
};

3、对外函数

threadpool\_t \*threadpool\_create( //创建线程池对象

int thread\_count, //线程池长度

int queue\_size, //任务队列长度

int flags); //是否是新创建的线程池

1、分配线程池内存空间

2、初始化threadpool\_t的成员变量，包括为线程数组和任务队列数组分配空间并初始化

3、调用pthread\_create创建thread\_count个线程，并在pool->thread[i]中存储tid，线程中执行threadpool\_thread函数。

int threadpool\_add( //向线程池任务队列中添加一个新任务

threadpool\_t \*pool, //线程池指针

void (\*routine)(void \*), //新任务函数指针  
void \*arg, //新任务参数

int flags); //无用

刚开始的时候任务队列中是没有任务的，这样当线程池创建线程时都会运行但是阻塞在条件变量上。假设线程池数组大小为thread\_count，这样就有thread\_count个运行的线程。

每次添加任务时，先找到添加的任务在任务队列的位置（(tail+1)%queue\_size），将任务加入到任务队列。任务队列tail+1。如果待执行的任务数量等于任务队列的长度，则无法添加。

每添加一个任务，就会唤醒一个线程执行任务，哪个线程竞争到互斥锁就由哪个线程执行。

线程执行完，线程退出。

如果先向任务队列中添加任务，这样创建线程就可以直接执行任务，执行的是任务队列头部的任务。之后head+1。待运行的任务减1.

int threadpool\_destroy( //销毁线程池

threadpool\_t \*pool, //线程池指针

int flags); //shutdown flags

4、辅助函数

static void \*threadpool\_thread(void \*threadpool);

//线程池中每个线程所执行的函数

int threadpool\_free(threadpool\_t \*pool);

//释放线程池内存