# 多线程

1 线程的概念

多线程有许多标准， 如：POSIX1、openmp 等。我们主要讲POSIX1的函数接口及标准。

在POSIX标准中pthread\_t为一个线程的标识符。进程有进程号PID 线程的“线程号”就是pthread\_t。pthread\_t类型在不同平台的定义不同，可能是int，可能是struct等，所以要用专用的函数来对pthread\_t类型进行处理。比如判断两个pthread\_t是否相等用：pthread\_equal() 函数。

2线程的创建

//创建一个线程

pthread\_creat:

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr,

void \*(\*start\_routine) (void \*), void \*arg);

pthread\_t \*thread, 新建线程的标识符

const pthread\_attr\_t \*attr, 规定创建线程的属性，空则表示默认属性创建。

void \*(\*start\_routine) (void \*), 线程要执行的函数指针

void \*arg 要执行函数的参数。

note:创建后的线程与main函数线程为同级的关系，执行mian函数的线程称为main线程。main线程和他创建出的线程谁先执行由调度决定。mian线程结束代表整个进程结束，所以main线程需要等待其他线程执行完再结束。

3 线程的退出

线程结束方式：

* 从线程要执行的函数中return
* 线程本身调用函数pthread\_exit
* 同进程的其他线程要求该线程结束

void pthread\_exit(void \*retval);

retval 为要返回的参数

线程结束的清理函数（结束钩子函数）：

pthread\_cleanup\_push

将函数挂入回调函数

pthread\_cleanup\_pop

决定回调函数是否执行

pthread\_cleanup\_push()/pthread\_cleanup\_pop()采用先入后出的栈结构管理（先挂入后执行），void routine(void \*arg)函数在调用pthread\_cleanup\_push()时压入清理函数栈，多次对pthread\_cleanup\_push()的调用将在清理函数栈中形成一个函数链，在执行该函数链时按照压栈的相反顺序弹出。execute参数表示执行到pthread\_cleanup\_pop()时是否在弹出清理函数的同时执行该函数，为0表示不执行，非0为执行；这个参数并不影响异常终止时清理函数的执行。

pthread\_cleanup\_push()/pthread\_cleanup\_pop()是以宏方式实现的。pthread\_cleanup\_push()带有一个"{"，而pthread\_cleanup\_pop()带有一个"}"，因此这两个函数必须成对出现，且必须位于程序的同一级别的代码段中才能通过编译。

pthread\_cleanup\_push();

...................

pthread\_exit(NULL);

...................

pthread\_cleanup\_pop();

这两句话之间任何导致该线程推出的语句都会触发回调函数执行,即使pthread\_cleanup\_pop()不会被执行到 也会触发

4 线程的回收

pthread\_join

//main线程等待其他线程--->wait()

int pthread\_join(pthread\_t thread, void \*\*retval);

pthread\_t thread, 要回收的线程

void \*\*retval 回收的状态

5 线程的取消: pthread\_cancel

pthread\_setcancelstate, 取消状态

pthread\_setcanceltype, 取消类型

pthread\_testcancel //设置一个取消点

POSIX默认线程取消点:会引发堵塞的系统调用

6 线程的分离:pthread\_detach

int pthread\_detach(pthread\_t thread);

7 互斥锁(1的信号量)

多个线程使用同一资源会相互竞争,要注意线程执行函数的值传递和地址传递

使用互斥锁可以避免竞争出现的错误

pthread\_mutex\_init()

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex,

const pthread\_mutexattr\_t \*restrict attr);

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

一种动态实现,一种静态实现

pthread\_mutex\_t \*restrict mutex, 初始化的锁的地址

const pthread\_mutexattr\_t \*restrict attr 锁的属性 NULL 为默认属性

//锁不上一直等 上锁

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex)

//解锁  
int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex)

//  
int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex)

//摧毁一个锁

pthread\_mutex\_destroy()

临界区的跳转注意要unlock