<https://www.cnblogs.com/luoxn28/p/6087649.html>

线程是程序中完成一个独立任务的完整执行序列，即一个可调度的实体；进程相当于运行中程序的一种抽象。根据运行环境的调度者的身份，线程可分为内核线程和用户线程。内核线程，在有的系统上称为LWP(Light Weight Process，轻量级线程)，运行在内核空间，由内核调度；用户线程运行在用户空间，由线程库来调度。当进程的一个内核线程获得CPU的使用权时，它就加载并运行一个用户线程。可见，内核线程相当于用户线程运行的‘容器’，一个进程可以拥有M个内核线程和N个用户线程，其中M<=N，并且一个系统的所有进程中，M和N的比值是固定的。

## 线程控制函数

**pthread\_create**

#include <pthread.h>

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

// 返回：成功返回0，出错返回错误编号

　　当pthread\_create函数返回成功时，有tidp指向的内存被设置为新创建线程的线程ID，其类型pthread\_t定义为：

#include <bits/pthreadtypes.h>

typedef unsigned long int pthread\_t;

　　attr参数用于设置各种不同的线程属性，为NULL时表示默认线程属性。新创建的线程从start\_rtn函数的地址开始运行，该函数只有一个无类型指针的参数arg，如果需要向start\_rtn函数传入的参数不止一个，可以把参数放入到一个结构中，然后把这个结构的地址作为arg的参数传入。

　　线程创建时并不能保证哪个线程会先运行：是新创建的线程还是调用线程。新创建的线程可以访问调用进程的地址空间，并且继承调用线程的浮点环境和信号屏蔽字，但是该线程的未决信号集被清除。那什么是未决信号呢，信号产生到信号被处理这段时间间隔，称信号是未决的。

**pthread\_exit**

#include <pthread.h>

void pthread\_exit(void \*rval\_ptr);

// 线程终止

　　线程在结束时最好调用该函数，以确保安全、干净的退出。pthread\_exit函数通过rval\_ptr参数向调用线程的回收者传递退出信息，进程中的其他线程可以调用pthread\_join函数访问到这个指针。pthread\_exit执行完后不会返回到调用者，而且永远不会失败。

线程可以通过以下三种方式退出，在不终止整个进程的情况下停止它的控制流：

* 线程只是从启动过程中退出，返回值是线程的退出码
* 线程可以被同一进程中的其他线程取消
* 线程调用pthread\_exit

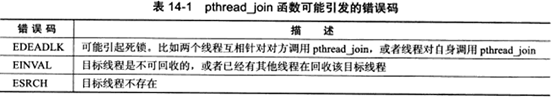
**pthread\_join**

#include <pthread.h>

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

// 返回：成功返回0，出错返回错误代码

　　thread是目标线程标识符，rval\_ptr指向目标线程返回时的退出信息，该函数会一直阻塞，直到被回收的线程结束为止。可能的错误码为：



**pthread\_cancel**

#include <pthread.h>

int pthread\_cancel(pthread\_t thread);

// 返回：成功返回0，出错返回错误代码

　　默认情况下，pthread\_cancel函数会使有thread标识的线程的表现为如同调用了参数为PTHREAD\_CANCEL的pthread\_exit函数，但是，接收到取消请求的目标线程可以决定是否允许被取消以及如何取消，这分别由以下两个函数来控制:

#include <pthread.h>

int pthread\_setcancelstate(int state, int \*oldstate);

int pthread\_setcanceltype(int type, int \*oldstate);

　　注意pthread\_cancel并不等待线程结束，它只是提出请求。

## 互斥量

　　互斥量本质是一把锁，在访问公共资源前对互斥量设置（加锁），确保同一时间只有一个线程访问数据，在访问完成后再释放（解锁）互斥量。在互斥量加锁之后，其他线程试图对该互斥量再次加锁时都会被阻塞，知道当前线程释放互斥锁。如果释放互斥量时有一个以上的互斥量，那么所有在该互斥量上阻塞的线程都会变成可运行状态，第一个变成运行的线程可以对互斥量加锁，其他线程看到互斥量依然是锁着的，只能再次阻塞等待该互斥量。

　　互斥量用pthread\_mutex\_t数据类型表示，在使用互斥量之前，必须使用pthread\_mutex\_init函数对它进行初始化，注意，使用完毕后需调用pthread\_mutex\_destroy。

#include <pthread.h>

int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*mutexattr);

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

// 两个函数返回值，成功返回0，否则返回错误码

　　pthread\_mutex\_init用于初始化互斥锁，mutexattr用于指定互斥锁的属性，若为NULL，则表示默认属性。除了用这个函数初始化互斥所外，还可以用如下方式初始化：pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER。  
　　pthread\_mutex\_destroy用于销毁互斥锁，以释放占用的内核资源，销毁一个已经加锁的互斥锁将导致不可预期的后果。

#include <pthread.h>

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

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

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

// 成功返回0，否则返回错误码

　　pthread\_mutex\_lock以原子操作给一个互斥锁加锁。如果目标互斥锁已经被加锁，则pthread\_mutex\_lock则被阻塞，直到该互斥锁占有者把它给解锁。  
　　pthread\_mutex\_trylock和pthread\_mutex\_lock类似，不过它始终立即返回，而不论被操作的互斥锁是否加锁，是pthread\_mutex\_lock的非阻塞版本。当目标互斥锁未被加锁时，pthread\_mutex\_trylock进行加锁操作；否则将返回EBUSY错误码。注意：这里讨论的pthread\_mutex\_lock和pthread\_mutex\_trylock是针对普通锁而言的，对于其他类型的锁，这两个加锁函数会有不同的行为。  
　　pthread\_mutex\_unlock以原子操作方式给一个互斥锁进行解锁操作。如果此时有其他线程正在等待这个互斥锁，则这些线程中的一个将获得它。

互斥锁使用示例：

[复制代码](javascript:void(0);)

/\*\*

\* 使用3个线程分别打印 A B C

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

pthread\_mutex\_t g\_mutex;

int g\_cnt = 0;

void \*func(void \*arg)

{

int loop = 3;

int result = (int)arg;

while (loop > 0) {

if (g\_cnt % 3 == result) {

switch (result)

{

case 0: {

printf("--- a\n");

break;

}

case 1: {

printf("--- b\n");

break;

}

case 2: {

printf("--- c\n");

break;

}

default: {

return NULL;

}

}

pthread\_mutex\_lock(&g\_mutex);

g\_cnt++;

loop--;

pthread\_mutex\_unlock(&g\_mutex);

}

}

return NULL;

}

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

{

pthread\_t t1, t2, t3;

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

pthread\_create(&t1, NULL, func, (void \*)0);

pthread\_create(&t2, NULL, func, (void \*)1);

pthread\_create(&t3, NULL, func, (void \*)2);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

pthread\_join(t3, NULL);

return 0;

}

[复制代码](javascript:void(0);)

## 读写锁

　　读写锁和互斥体类似，不过读写锁有更高的并行性，互斥体要么是锁住状态，要么是不加锁状态，而且一次只有一个线程可以对其加锁。而读写锁可以有3个状态，读模式下锁住状态，写模式下锁住状态，不加锁状态。一次只有一个线程可以占有写模式的读写锁，但是多个线程可以同时占用读模式的读写锁。读写锁适合对数据结构读的次数远大于写的情况。

　　当读写锁是写加锁状态时，在这个锁被解锁之前，所有试图对这个锁加锁的线程都会被阻塞。当读写锁是读加锁状态时，所有试图以读模式对它进行加锁的线程都可以得到访问权，但是任何希望以写模式对此锁进行加锁的线程都会阻塞，直到所有的线程释放它们的读锁为止。

#include<pthread.h>

int pthread\_rwlock\_init(pthread\_rwlock\_t \*restrict rwlock, const pthread\_rwlockaddr\_t \*restrict attr);

int pthread\_rwlock\_destroy(pthread\_rwlock\_t \*restrict rwlock);

// 成功返回0，否则返回错误码

　　通过pthread\_rwlock\_init初始化读写锁，如果希望读写锁有默认属性，可以传一个NULL指针给attr。当不再需要读写锁时，调用pthread\_rwlock\_destroy做清理工作。

#include<pthread.h>

int pthread\_rwlock\_rdlock(pthread\_rwlock\_t \*restrict rwlock);

int pthread\_rwlock\_wrlock(pthread\_rwlock\_t \*restrict rwlock);

int pthread\_rwlock\_unlock(pthread\_rwlock\_t \*restrict rwlock);

// 成功返回0，否则返回错误码

　　读写锁的读加锁、写加锁和解锁操作。

读写锁程序示例：

[复制代码](javascript:void(0);)

/\*\*

\* 两个读线程读取数据，一个写线程更新数据

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

#define READ\_THREAD 0

#define WRITE\_THREAD 1

int g\_data = 0;

pthread\_rwlock\_t g\_rwlock;

void \*func(void \*pdata)

{

int data = (int)pdata;

while (1) {

if (READ\_THREAD == data) {

pthread\_rwlock\_rdlock(&g\_rwlock);

printf("-----%d------ %d\n", pthread\_self(), g\_data);

sleep(1);

pthread\_rwlock\_unlock(&g\_rwlock);

sleep(1);

}

else {

pthread\_rwlock\_wrlock(&g\_rwlock);

g\_data++;

printf("add the g\_data\n");

pthread\_rwlock\_unlock(&g\_rwlock);

sleep(1);

}

}

return NULL;

}

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

{

pthread\_t t1, t2, t3;

pthread\_rwlock\_init(&g\_rwlock, NULL);

pthread\_create(&t1, NULL, func, (void \*)READ\_THREAD);

pthread\_create(&t2, NULL, func, (void \*)READ\_THREAD);

pthread\_create(&t3, NULL, func, (void \*)WRITE\_THREAD);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

pthread\_join(t3, NULL);

pthread\_rwlock\_destroy(&g\_rwlock);

return 0;

}

[复制代码](javascript:void(0);)

## 条件变量

　　条件变量是线程可用的一种同步机制，条件变量给多个线程提供了一个回合的场所，条件变量和互斥量一起使用，允许线程以无竞争的方式等待特定的条件发生。条件变量本事是由互斥体保护的，线程在改变条件状态之前必须首先锁住互斥量，其他线程在获取互斥量之前就不会觉察到这种变化，因为互斥量必须锁定之后才改变条件。

#include<pthread.h>

pthread\_cond\_init(pthread\_cond\_t \*restrict cond, const pthread\_condattr\_t \*restrict attr);

pthread\_cond\_destroy(pthread\_cont\_t \*cond);

// 成功返回0，否则返回错误码

　　使用条件变量前调用pthread\_cond\_init初始化，使用完毕后调用pthread\_cond\_destroy做清理工作。除非需要创建一个具有非默认属性的条件变量，否则pthread\_cond\_init函数的attr参数可以设置为NULL。

#include<pthread.h>

int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);

int pthread\_cond\_signal(pthread\_cond\_t \*cond);

int pthread\_cond\_wait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex);

// 成功返回0，否则返回错误码

　　传递给pthread\_cond\_wait的互斥量对条件进行保护，调用者把锁住互斥量传给函数，函数然后自动把调用线程放到等待条件的线程列表上，对互斥量解锁。这就关闭了条件检查和线程进入休眠状态等待条件改变这两个操作之间的时间通道，这样线程就不会错过条件的任何变化。pthread\_cond\_wait函数返回时，互斥量再次被锁住。

　　pthread\_cond\_broadcast用广播的形式唤醒所有等待条件变量的线程。pthread\_cond\_signal用于唤醒一个等待条件变量的线程，至于哪个线程被唤醒，取决于线程的优先级和调度机制。有时候需要唤醒一个指定的线程，但pthread没有对该需要提供解决方法。可以间接实现该需求：定义一个能够唯一表示目标线程的全局变量，在唤醒等待条件变量的线程前先设置该变量为目标线程，然后以广播形式唤醒所有等待条件变量的线程，这些线程被唤醒后都检查改变量是否是自己，如果是就开始执行后续代码，否则继续等待。

条件变量程序示例：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define err\_sys(msg) \

do { perror(msg); exit(-1); } while(0)

#define err\_exit(msg) \

do { fprintf(stderr, msg); exit(-1); } while(0)

pthread\_cond\_t cond;

void \*r1(void \*arg)

{

pthread\_mutex\_t\* mutex = (pthread\_mutex\_t \*)arg;

static int cnt = 10;

while(cnt--)

{

printf("r1: I am wait.\n");

pthread\_mutex\_lock(mutex);

pthread\_cond\_wait(&cond, mutex); /\* mutex参数用来保护条件变量的互斥锁，调用pthread\_cond\_wait前mutex必须加锁 \*/

pthread\_mutex\_unlock(mutex);

}

return "r1 over";

}

void \*r2(void \*arg)

{

pthread\_mutex\_t\* mutex = (pthread\_mutex\_t \*)arg;

static int cnt = 10;

while(cnt--)

{

//pthread\_mutex\_lock(mutex); //这个地方不用加锁操作就行

printf("r2: I am send the cond signal.\n");

pthread\_cond\_signal(&cond);

//pthread\_mutex\_unlock(mutex);

sleep(1);

}

return "r2 over";

}

int main(void)

{

pthread\_mutex\_t mutex;

pthread\_t t1, t2;

char\* p1 = NULL;

char\* p2 = NULL;

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

pthread\_cond\_init(&cond, NULL);

pthread\_create(&t1, NULL, r1, &mutex);

pthread\_create(&t2, NULL, r2, &mutex);

pthread\_join(t1, (void \*\*)&p1);

pthread\_join(t2, (void \*\*)&p2);

pthread\_cond\_destroy(&cond);

pthread\_mutex\_destroy(&mutex);

printf("s1: %s\n", p1);

printf("s2: %s\n", p2);

return 0;

}

[复制代码](javascript:void(0);)

## 自旋锁

　　自旋锁和互斥量类似，但它不是通过休眠使进程阻塞，而是在获取锁之前一直处于忙等（自旋）状态，自旋锁可用于下面的情况：锁被持有的时间短，并且线程不希望再重新调度上花费太多的成本。自旋锁通常作为底层原语用于实现其他类型的锁。根据他们所基于的系统架构，可以通过使用测试并设置指令有效地实现。当然这里说的有效也还是会导致CPU资源的浪费：当线程自旋锁变为可用时，CPU不能做其他任何事情，这也是自旋锁只能够被只有一小段时间的原因。

#include <pthread.h>

int pthread\_spin\_init(pthread\_spinlock\_t \*lock, int pshared);

int pthread\_spin\_destroy(pthread\_spinlock\_t \*lock);

　　pshared参数表示进程共享属性，表明自旋锁是如何获取的，如果它设为PTHREAD\_PROCESS\_SHARED，则自旋锁能被可以访问锁底层内存的线程所获取，即使那些线程属于不同的进程。否则pshared参数设为PTHREAD\_PROCESS\_PROVATE，自旋锁就只能被初始化该锁的进程内部的线程访问到。

#include <pthread.h>

int pthread\_spin\_lock(pthread\_spinlock\_t \*lock);

int pthread\_spin\_trylock(pthread\_spinlock\_t \*lock);

int pthread\_spin\_unlock(pthread\_spinlock\_t \*lock);

　　如果自旋锁当前在解锁状态，pthread\_spin\_lock函数不要自旋就可以对它加锁，试图对没有加锁的自旋锁进行解锁，结果是未定义的。需要注意，不要在持有自旋锁情况下可能会进入休眠状态的函数，如果调用了这些函数，会浪费CPU资源，其他线程需要获取自旋锁需要等待的时间更长了。

自旋锁使用示例：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

pthread\_spinlock\_t g\_lock;

int g\_data = 0;

void \*func(void \*arg)

{

while (1) {

pthread\_spin\_lock(&g\_lock);

g\_data++;

printf("----------- %d\n", g\_data);

sleep(1);

pthread\_spin\_unlock(&g\_lock);

}

}

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

{

pthread\_t tid;

pthread\_spin\_init(&g\_lock, PTHREAD\_PROCESS\_PRIVATE);

pthread\_create(&tid, NULL, func, NULL);

pthread\_create(&tid, NULL, func, NULL);

pthread\_create(&tid, NULL, func, NULL);

pthread\_join(tid, NULL);

return 0;

}

[复制代码](javascript:void(0);)

## 屏障

　　屏障是用户协调多个线程并行工作的同步机制，屏障允许每个线程等待，直到所有合作的线程都到达某一点，然后从该点出继续执行。pthread\_join其实就是一种屏障，允许一个线程等待，直到另一个线程退出。但是屏障对象的概念更广，它们允许任意数量的线程等待，直到所有的线程完成处理工作，而线程不需要退出，所有线程达到屏障后可以继续工作。

#include <pthread.h>

int pthread\_barrier\_init(pthread\_barrier\_t \*restrict barrier, const pthread\_barrierattr\_t \*restrict attr, unsigned int count);

int pthread\_barrier\_destroy(pthread\_barrier\_t \*barrier);

// 成功返回0，否则返回错误编号

　　初始化屏障时，可以使用count参数指定，在允许所有线程继续运行前，必须达到屏障的线程数目。attr指定屏障属性，NULL为默认属性。

#include <pthread.h>

int pthread\_barrier\_wait(pthread\_barrier\_t \*barrier);

// 成功返回0，否则返回错误编号

　　可以使用pthread\_barrier\_wait函数来表明，线程已完成工作，准备等所有其他线程赶过来。调用pthread\_barrier\_wait的线程在屏障计数未满足条件时，会进入休眠状态。如果该线程是最后一个调用pthread\_barrier\_wait的线程，则所有的线程会被唤醒。

　　一旦到达屏障计数值，而且线程处于非阻塞状态，屏障就可以被重复使用。

屏障使用示例：

[复制代码](javascript:void(0);)

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

pthread\_barrier\_t g\_barrier;

void \*func(void \*arg)

{

int id = (int )arg;

if (id == 0) {

printf("thread 0\n");

sleep(1);

pthread\_barrier\_wait(&g\_barrier);

printf("thread 0 come...\n");

}

else if (id == 1) {

printf("thread 1\n");

sleep(2);

pthread\_barrier\_wait(&g\_barrier);

printf("thread 1 come...\n");

}

else if (id == 2) {

printf("thread 2\n");

sleep(3);

pthread\_barrier\_wait(&g\_barrier);

printf("thread 2 come...\n");

}

return NULL;

}

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

{

pthread\_t t1, t2, t3;

pthread\_barrier\_init(&g\_barrier, NULL, 3);

pthread\_create(&t1, NULL, func, (void \*)0);

pthread\_create(&t2, NULL, func, (void \*)1);

pthread\_create(&t3, NULL, func, (void \*)2);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

pthread\_join(t3, NULL);

return 0;

}

[复制代码](javascript:void(0);)

参考：

　　1、《UNIX环境高级编程 第三版》线程章节

　　2、[ThinkInTechnology](https://github.com/luoxn28/ThinkInTechnology)