threads函数库的简介

介绍

threads函数库是c11标准新加的多线程函数库，使用时，要包含threads.h头文件，编译时规定std=c11
我们都知道，程序运行时会产生进程，程序会被加载到进程的地址空间中。但是，进程还要有执行单位，那就是线程。进程创建时，肯定要有至少一个线程执行代码。进程刚创建时创建的线程叫主线程，主线程创建的线程叫子线程。
glibc在内的大部分标准库都不支持使用这个函数库，不支持这个函数库的编译器会定义__STDC_NO_THREADS__宏。而musl的标准库支持这个函数库。因此，如无特殊说明，本文使用的编译器均是musl-gcc。

musl的安装与使用

musl的安装

在ubuntu中，我们可以采用apt安装这个标准库。在终端中输入：

```
sudo apt install musl
sudo apt install musl-tools
```

就可以安装musl标准库了。
我们可以输入：

```
musl-gcc --version
```

检测是否安装成功。musl-gcc是采用musl函数库的gcc编译器，如果安装成功，会看到：

```
gcc (Ubuntu 5.5.0-12ubuntu1~16.04) 5.5.0 20171010
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

musl的使用

使用这个标准库非常简单，只需要输入：

```
musl-gcc ...
```

就可以了。使用的方法类似普通gcc的使用。

线程的使用

线程的创建

线程的创建采用thrd_create(3)函数，这个函数的原形如下：

```
int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
```

这个函数将会创建一个运行func函数的线程，在thrd_create(3)函数里，arg将会作为func函数的参数。如果线程创建成功，thr指向的对象将被设置为所创建线程的标识符。这个函数完成的同时，所创建的线程将开始。

以下是对其参数的详细解释：

- thr: 指向将要存放线程标识符的地址
- func: 需要执行的函数。注意：这个函数是int (*)(void *)类型的，这意味着函数的返回值是int类型，接受一个void *类型的参数。
- arg: 向执行函数传递的参数

这个函数的返回值是一个枚举，其定义如下：

```
enum {

    thrd_success = /* unspecified */,
    thrd_nomem = /* unspecified */,
    thrd_timedout = /* unspecified */,
    thrd_busy = /* unspecified */,
    thrd_error = /* unspecified */

};
```

其中，thrd_success是线程创建成功的返回值。thrd_nomem表示因地址空间不足造成线程创建失败。thrd_error表示其他错误发生。这个枚举类型的其他枚举形式不会返回。
另外，当线程结束、加入或脱离后，线程标识符可能被重新使用。
当线程返回，或调用thrd_exit(3)时，线程结束。线程的结束码会被设为执行函数的返回值或thrd_exit(3)函数的参数。

线程的加入与脱离

当线程结束、加入或脱离后，线程标识符可能被重新使用。那么，什么是加入、脱离呢？如何操作？

线程的加入

线程的加入意味着阻塞当前的线程直到线程结束。通过thrd_join(3)函数可以实现线程的加入。

```
int thrd_join( thrd_t thr, int *res );
```

这个函数与线程同时终止，其参数如下：

- thr：要进行加入操作的线程
- res：当线程结束时，将线程的结束码放入该参数指向的地址中。这个参数可以为NULL。

这个函数的返回值仍然是上文所说的枚举。如果成功，则返回thrd_success，否则返回thrd_error。

线程的脱离

线程的脱离与加入相反，意味着将线程同当前环境脱离出来，当线程结束时，将自动销毁，而不会阻塞线程。通过thrd_detach(3)实现线程脱离。

```c
int thrd_detach( thrd_t thr );
```

这个函数的参数只有一个，即thr，表示要进行脱离操作的线程。这个函数的返回值仍然是上文所说的枚举。如果成功，则返回thrd_success，否则返回thrd_error。

线程的终止

在以下情况下，线程终止：

- 执行函数调用**thrd_exit(3)**函数；
- 执行函数返回，这相当于调用**thrd_exit(3)**函数；
- 本进程中任何线程调用**exit(3)**函数，或main函数返回，这将会使本进程所有线程终止。

接下来给大家介绍**thrd_exit(3)**函数，在C11标准中，其原型如下：

```c
_Noreturn void thrd_exit( int res );
```

在C23标准中，其原型如下：

```c
[[noreturn]] void thrd_exit( int res );
```

res表示结束码，这将会成为thrd_join(3)的返回值。
exit(3)是标准库中的函数，这里也给大家介绍一下：

```c
void exit(int status);
```

使用时需要包含**stdlib.h**头文件。
这个函数非常简单。它会导致进程终止。并且**status**参数与上**0377**的结果将会返回给父进程。为什么要要与上一个0377？因为子进程的状态改变不只有终止，还有被信号停止、被信号重新开始。为了使系统及用户区分，因此需要进行位运算。
事实上，exit(0)等价于main函数中return 0.这说明main函数返回时也会与上**0377**.所以，如果不知道位操作，可以简单的把exit看成main函数的返回。

举个例子

以下程序展示了如何创建一个线程，输出Hello World：

```c
#include <stdio.h>

#ifndef __STDC_NO_THREADS__
#include <threads.h>
#endif

int func(void *args){
    printf("Hello World\n");
    return 0;
}

int main(void){
    #ifdef __STDC_NO_THREADS__
    printf("cannot use threads\n");
    #else
    thrd_t th;
    thrd_create(&th, func, NULL);
    thrd_join(th, NULL);
    #endif
    return 0;
}
```

如果__STDC_NO_THREADS__宏被定义，则这个程序的流程如下：

1. 定义函数 func：
    - 这个函数打印"Hello World"并返回0。
2. main 函数：
    - 定义一个线程变量th。
    - 使用thrd_create函数创建一个新线程，执行函数是func，并且没有传递任何参数。
    - 使用thrd_join进行线程的加入操作，等待线程完成。
3. 返回0，结束程序。

这个程序对__STDC_NO_THREADS__宏进行了多次检测，如果没有这个宏，就不能使用threads库。这样的写法是为了更好的兼容跨平台程序。

我们使用musl编译：

```
musl-gcc -std=c11 th1.c
./a.out
```

输出：

```
Hello World
```

可以看到程序成功输出了Hello World。使用threads库可以更好的支持跨平台应用。
如果我们把程序的thrd_join改成thrd_detach：

```c
#include <stdio.h>

#ifndef __STDC_NO_THREADS__
#include <threads.h>
#endif

int func(void *args){
    printf("Hello World\n");
    return 0;
}

int main(void){
    #ifdef __STDC_NO_THREADS__
    printf("cannot use threads\n");
    #else
    thrd_t th;
    thrd_create(&th, func, NULL);
    thrd_detach(th);
    #endif
    return 0;
}
```

则不会有输出。这是因为线程创建后main函数就返回了，线程终止。

互斥体

互斥体又叫互斥锁，可以解决多个线程同时访问一块地址的问题。先看一段代码：

```c
#include <threads.h>
#include <stdio.h>

int test = 10;

int func(void *args){
    while(--test > 0){
        printf("%d\n", test);
    }
    return 0;
}

int main(void){
    thrd_t t1, t2;
    thrd_create(&t1, func, NULL);
    thrd_create(&t2, func, NULL);
    thrd_join(t1, NULL);
    thrd_join(t2, NULL);
    printf("%d\n", test);
    return 0;
}
```

这段代码的流程如下：

1. 定义一个全局变量test，并初始化为10。
2. 执行函数定义：
    - 将test变量递减到0。每次循环，它都会打印当前的test值，然后继续循环。
3. 主函数：
    - 创建一个新线程，执行函数是func。线程的标识符是t1。
    - 同样地，创建另一个新线程，执行函数是func。线程的标识符是t2。
    - 对两个线程进行加入操作，等待完成。
    - 打印全局变量test的值。。

这段代码创建了两个线程，以减少test变量到0。我们来看它的输出结果：

```
8
7
6
5
4
3
2
1
9
-1
```

可以看到，它最后的结果是-1，不是我们想要的0。这是因为两个线程同时操作test变量造成的。

如何解决这个问题？我们可以使用互斥体：

互斥体的创建

互斥体的创建使用mtx_init(3)函数，其原型如下：

```
int mtx_init( mtx_t* mutex, int type );
```

其参数的解释如下：

- mutex：指向的对象将被设置为互斥体的标识符
- type：互斥体的类型。互斥体有以下类型：
  - mtx_plain：简单的不递归互斥体
  - mtx_timed：支持超时的不递归互斥体
  - mtx_plain | mtx_recursive：简单的递归互斥体
  - mtx_timed | mtx_recursive：支持超时的不递归互斥体

其返回值仍然是上文所说的枚举。如果成功，则返回thrd_success，否则返回thrd_error。

互斥体的使用

互斥体可以进行锁定和解锁操作

互斥体的锁定

互斥体的锁定采用mtx_lock(3)函数，其原型如下：

```
int mtx_lock( mtx_t* mutex );
```

它将会阻塞当前线程直到成功锁定，如果互斥体已经被锁定且互斥体不递归，则不会进行任何行为，只会阻塞线程，直到互斥体被解锁，然后再锁定。mutex指向要操作的互斥体。其返回值仍然是上文所说的枚举。如果成功，则返回thrd_success，否则返回thrd_error。
互斥体的锁定还可以采用mtx_trylock(3)函数，其原型如下：

```
int mtx_trylock( mtx_t *mutex );
```

它将会尝试锁定互斥体而不阻塞线程，如果互斥体已被锁定则立刻返回。mutex指向要操作的互斥体。其返回值仍然是上文所说的枚举。如果成功，则返回thrd_success，如果出现错误返回thrd_error，但是，如果互斥体已经被锁定则返回thrd_busy。

互斥体的解锁

互斥体的解锁采用mtx_unlock(3)函数：

```
int mtx_unlock( mtx_t *mutex );
```

他将会解锁互斥体，如果互斥体已被解锁，则不会进行任何操作。互斥体解锁之后，被mtx_lock(3)函数阻塞的线程将不再被阻塞。mutex指向要操作的互斥体。其返回值仍然是上文所说的枚举。如果成功，则返回thrd_success，否则返回thrd_error。

互斥体的销毁

互斥体的销毁采用mtx_destroy(3)函数，其原型如下：

```
void mtx_destroy( mtx_t *mutex );
```

它将销毁mutex指向的互斥体，如果有线程正在这个互斥体上等待，则不会进行任何操作。它没有返回值，因此需要注意销毁时应确保没有线程在互斥体上等待。

互斥体类型

互斥体的类型被定义为一个枚举：

```
enum {
    mtx_plain = /* unspecified */,
    mtx_recursive = /* unspecified */,
    mtx_timed = /* unspecified */
};
```

分别表示平常互斥体，递归互斥体，定时互斥体。

平常互斥体

平常互斥体就是普通的互斥体，支持锁定和解锁等操作。同一时间只能被锁定一次

我们可以把前面的代码进行一定的修改：

```c
#include <threads.h>
#include <stdio.h>

int test = 10;
mtx_t m;

int func(void *args){
    while(test > 0){
        mtx_lock(&m);
        test--;
        mtx_unlock(&m);
        printf("%d\n", test);
    }
    return 0;
}

int main(void){
    thrd_t t1, t2;
    mtx_init(&m, mtx_plain);
    thrd_create(&t1, func, NULL);
    thrd_create(&t2, func, NULL);
    thrd_join(t1, NULL);
    thrd_join(t2, NULL);
    mtx_destroy(&m);
    printf("%d\n", test);
    return 0;
}
```

这段代码在操作test之前对互斥体m进行锁定操作，并在操作之后进行解锁。

可以看到，其运行结果为：

```
9
8
7
6
5
4
3
2
1
0
0
```

因此，我们利用互斥体，成功的防止的线程同时操作变量带来的冲突。

死锁

死锁是平常互斥体当中的重要问题，我们来看段代码：

```c
#include <stdio.h>
#include <threads.h>
#include <unistd.h>

mtx_t m1, m2;

int th1(void *args){
    // 用睡眠模拟线程的操作
    mtx_lock(&m1);
    printf("th1 locked m1\n");
    usleep(200);
    mtx_lock(&m2);
    printf("th1 locked m2\n");
    mtx_unlock(&m1);
    printf("th1 unlocked m1\n");
    usleep(200);
    mtx_unlock(&m2);
    printf("th1 unlocked m2\n");
    return 0;
}

int th2(void *args){
    mtx_lock(&m2);
    printf("th2 locked m2\n");
    usleep(200);
    mtx_lock(&m1);
    printf("th2 locked m1\n");
    mtx_unlock(&m2);
    printf("th2 unlocked m2\n");
    usleep(200);
    mtx_unlock(&m1);
    printf("th2 unlocked m1\n");
    return 0;
}

int main(void){
    thrd_t t1, t2;
    mtx_init(&m1, mtx_plain);
    mtx_init(&m2, mtx_plain);
```

```
    thrd_create(&t1, th1, NULL);
    thrd_create(&t2, th2, NULL);
    thrd_join(t1, NULL);
    thrd_join(t2, NULL);
    return 0;
}
```

以下是这段代码的流程：

1. 初始化两个互斥体m1和m2，它们被设置为平常互斥体。
2. 创建两个线程t1和t2，分别执行th1和th2函数。
3. 在th1函数中：
    - 首先锁定m1，此时打印"th1 locked m1"。
    - 等待200微秒。
    - 锁定m2，此时打印"th1 locked m2"。
    - 解锁m1，此时打印"th1 unlocked m1"。
    - 等待200微秒。
    - 解锁m2，此时打印"th1 unlocked m2"。
4. 在th2函数中：
    - 首先锁定m2，此时打印"th2 locked m2"。
    - 等待200微秒。
    - 锁定m1，此时打印"th2 locked m1"。
    - 解锁m2，此时打印"th2 unlocked m2"。
    - 等待200微秒。
    - 解锁m1，此时打印"th2 unlocked m1"。
5. 通过thrd_join等待两个线程结束。

运行结果：

```
th2 locked m2
th1 locked m1
（然后进入死循环）
```

为什么会进入死循环呢？因为线程th1需要在锁定m2后解锁m1，线程th2需要在锁定m1后解锁m2，这导致了“A拿到A锁，想要拿B锁，B拿着B锁，想要A锁”的死锁情况。

递归互斥体

递归互斥体允许（同一个线程）进行多次锁定，只有解锁的次数与锁定的次数一样多，互斥体才会被真正解锁。
递归互斥体可以解决死锁的问题，如以下代码所示：

```c
#include <stdio.h>
#include <threads.h>
#include <unistd.h>

mtx_t m;

int th1(void *args){
    // 用睡眠模拟线程的操作
    // 这里为了区分，仍采用m1，m2作为输出名称
    mtx_lock(&m);
    printf("th1 locked m1\n");
    usleep(200);
    mtx_lock(&m);
    printf("th1 locked m2\n");
    mtx_unlock(&m);
    printf("th1 unlocked m1\n");
    usleep(200);
    mtx_unlock(&m);
    printf("th1 unlocked m2\n");
    return 0;
}

int th2(void *args){
    mtx_lock(&m);
    printf("th2 locked m2\n");
    usleep(200);
    mtx_lock(&m);
    printf("th2 locked m1\n");
    mtx_unlock(&m);
    printf("th2 unlocked m2\n");
    usleep(200);
    mtx_unlock(&m);
    printf("th2 unlocked m1\n");
    return 0;
}

int main(void){
    thrd_t t1, t2;
    mtx_init(&m, mtx_plain | mtx_recursive);
```

```
    thrd_create(&t1, th1, NULL);
    thrd_create(&t2, th2, NULL);
    thrd_join(t1, NULL);
    thrd_join(t2, NULL);
    return 0;
}
```

我们把m1和m2改成了同一个递归互斥体m，避免了死锁的问题。
输出结果如下：

```
th1 locked m1
th1 locked m2
th1 unlocked m1
th1 unlocked m2
th2 locked m2
th2 locked m1
th2 unlocked m2
th2 unlocked m1
```

可以看到，程序不再陷入死循环，而是成功的输出了结果。这说明，递归互斥体可以解决死锁问题。

定时互斥体

定时互斥体允许在锁定时判断是否超时，如果超时就会停止阻塞线程。

定时互斥体的锁定

可以采用平常互斥体的锁定方法，也可以采用mtx_timedlock(3)函数对定时互斥体进行锁定操作：

```
int mtx_timedlock( mtx_t *restrict mutex,
                   const struct timespec *restrict time_point );
```

其中，mutex为一个定时互斥体，如果不是一个定时互斥体，则不会进行任何操作。time_point指向要等待到的绝对日历时间，是timespec类型的。如果成功，则返回thrd_success；若超时，则返回thrd_timedout；否则返回thrd_error。

timespec结构体

timespec结构体将时间拆分为秒和纳秒，它定义在time.h头文件中，有两个成员变量：

```
time_t tv_sec;
long tv_nsec;
```

注意：这两个成员变量的声明顺序未指定，标准库可能添加了其他成员变量。
我们可以通过**timespec_get(3)**函数获取当前的时间，其原型如下：

```
int timespec_get( struct timespec *ts, int base );
```

其中，ts是指向timespec结构体的指针，base是时间基底，可以为TIME_UTC或其他指示时间基底的任意非零整数值。但是，一般情况下使用TIME_UTC，因为定时互斥体要求使用以TIME_UTC为基底的时间。
在定时互斥体的使用中，一般会传入一个离当前时间一定距离的时刻：

```
struct timespec now_time;
timespec_get(&now_time, TIME_UTC);
now_time.tv_sec += 1;
```

接下来只需要向mtx_timedlock(3)传入now_time的地址即可：

```
mtx_timedlock(&m, &now_time);
```

定时互斥体的使用

我们来看一下这段代码：

```c
#include <stdio.h>
#include <threads.h>
#include <time.h>
#include <unistd.h>

mtx_t m;

int th1(void *args){
    usleep(200);
    struct timespec now_time;
    timespec_get(&now_time, TIME_UTC);
    now_time.tv_sec += 1;
    printf("th1 will lock m\n");
    mtx_timedlock(&m, &now_time);
    printf("th1 locked m\n");
    return 0;
}

int th2(void *args){
    mtx_lock(&m);
    printf("th2 locked m\n");
    sleep(2);
    mtx_unlock(&m);
    printf("th2 unlocked m\n");
}

int main(void){
    thrd_t t1, t2;
    mtx_init(&m, mtx_timed);
    thrd_create(&t1, th1, NULL);
    thrd_create(&t2, th2, NULL);
    thrd_join(t1, NULL);
    thrd_join(t2, NULL);
    return 0;
}

```

这段代码的流程如下：

1. 首先，初始化一个互斥体m，它被设置为定时互斥体。

2. 创建两个线程t1和t2，分别执行th1和th2函数。
3. 在th1函数中：
    - 线程th1首先休眠200微秒。
    - 获取当前时间并设置为下1秒后。
    - 打印"th1 will lock m"，然后定时锁定互斥体。
    - 如果定时锁定不再阻塞线程，则打印"th1 locked m"。
4. 在th2函数中：
    - 线程th2直接锁定互斥体m，并打印"th2 locked m"。
    - 然后，线程th2休眠2秒。
    - 解锁互斥体m，并打印"th2 unlocked m"。
5. 通过thrd_join等待两个线程结束。

这个程序创建了两个线程，其中一个线程定时锁定互斥体，另一个普通锁定互斥体，最终的输出如下：

```
th2 locked m
th1 will lock m
（一秒后）
th1 locked m
（一秒后）
th2 unlocked m
```

可以看到，当阻塞超时后，th1线程直接停止了阻塞，继续进行了下面的代码，1秒后，th2线程才解锁。这样的机制也可以防止死锁，但需要注意不能多个同时操作同一数据。

条件变量

条件变量允许一个线程唤醒在条件变量上被阻塞的线程，与互斥锁有些相似之处。一般情况下，我们会在一个条件满足之后，再唤醒另一个线程。比如1号线程需要完成一个步骤后，才能让2号线程执行。

条件变量的创建

条件变量的创建使用cnd_init(3)函数，其原型如下：

```
int cnd_init( cnd_t* cond );
```

这个函数将会初始化条件变量，并将cond指向的对象设为其条件变量的标识符。若成功创建则返回thrd_success，若地址空间不足则返回thrd_nomem，若出现其他错误则返回thrd_error。

条件变量的使用

条件变量可以进行阻塞和唤醒操作。

条件变量的阻塞

条件变量的阻塞用**cnd_wait(3)**函数，其原型如下：

```
int cnd_wait( cnd_t* cond, mtx_t* mutex );
```

它将会依次执行以下步骤：

1. 解锁mutex指向的互斥体
2. 阻塞当前线程，直到cond指向的条件变量被唤醒
3. 锁定mutex指向的互斥体，这意味着如果mutex已经被锁定，则会继续阻塞当前线程

若成功创建则返回thrd_success，若地址空间不足则返回thrd_nomem，若出现其他错误则返回thrd_error。

还可以定时地进行阻塞，使用**cnd_timedwait(3)**函数，其原型如下：

```
int cnd_timedwait( cnd_t* restrict cond, mtx_t* restrict mutex,
                   const struct timespec* restrict time_point );
```

其用法与定时互斥体类似，并会像cnd_wait(3)函数一样执行。如果成功，则返回thrd_success；若超时，则返回thrd_timedout；否则返回thrd_error。

条件变量的唤醒

条件变量的唤醒实际上就是停止条件变量的阻塞，条件变量可以唤醒一个或所有线程。

唤醒一个线程

cnd_signal(3)可以唤醒一个线程，其原型如下：

```
int cnd_signal( cnd_t *cond );
```

它将会唤醒cond指向的条件变量上的一个线程，如无线程被阻塞，则不进行操作并返回thrd_success。如果成功，则返回thrd_success，否则返回thrd_error。

唤醒所有线程

唤醒所有线程用cnd_broadcast(3)函数，其原型如下：

```
int cnd_broadcast( cnd_t *cond );
```

它将会唤醒cond指向的条件变量上的所有线程，如无线程被阻塞，则不进行操作并返回thrd_success。如果成功，则返回thrd_success，否则返回thrd_error。

条件变量的销毁

条件变量的销毁采用cnd_destroy(3)函数，其原型如下：

```
void cnd_destroy( cnd_t* cond );
```

它将销毁cond指向的条件变量，如果有线程正在这个条件变量上等待，则不会进行任何操作。它没有返回值，因此需要注意销毁时应确保没有线程在条件变量上等待。

举个例子

我们来看以下代码：

```c
#include <stdio.h>
#include <stdlib.h>
#include <threads.h>
#include <unistd.h>

mtx_t mwait;
cnd_t cwait;
thrd_t th;

int thread_deal_fn(void *arg) {
    mtx_lock(&mwait);
    printf("thread_deal_fn 睡眠 1s\n");
    sleep(1);
    printf("thread_deal_fn 唤醒 main 线程\n");
    cnd_broadcast(&cwait);
    sleep(1);
    mtx_unlock(&mwait);
    return 0;
}

int th1(void *arg){
    mtx_lock(&mwait);
    return 0;
}

int main(){
    // 初始化
    mtx_init(&mwait, mtx_plain);
    cnd_init(&cwait);

    thrd_t t1;
    thrd_create(&t1, th1, NULL);
    thrd_join(t1, NULL);

    thrd_create(&th, thread_deal_fn, NULL);

    printf("main 线程阻塞\n");
    cnd_wait(&cwait, &mwait);

```

```
    printf("main 线程被唤醒，准备退出\n");
    return 0;
}
```

这段代码的流程如下：

1. 初始化互斥体mwait为平常互斥体。
2. 初始化条件变量cwait。
3. 创建线程th1并执行。该线程锁定mwait互斥体，然后立即返回。
4. 等待线程th1结束。
5. 创建线程thread_deal_fn并执行：
    - 首先，该线程锁定mwait互斥体。
    - 然后，线程休眠1秒。
    - 打印"thread_deal_fn 唤醒 main 线程"。
    - 唤醒所有等待在条件变量cwait上的线程。
    - 再次休眠1秒。
    - 最后，解锁mwait互斥体。
6. 在主线程中：
    - 先打印"main 线程阻塞"
    - 在条件变量cwait上等待。
7. 当主线程被唤醒后，它会打印"main 线程被唤醒，准备退出"，然后程序结束。

运行结果：

```
main 线程阻塞
thread_deal_fn 睡眠 1s
（一秒后）
thread_deal_fn 唤醒 main 线程
（一秒后）
main 线程被唤醒，准备退出
```

th1线程在thread_deal_fn线程之前执行，就把mwait锁定，但执行cnd_wait后，thread_deal_fn线程中的锁定指令没有阻塞线程，而是成功锁定，因此，cnd_wait函数会解锁互斥体。main线程阻塞1秒后，thread_deal_fn 唤醒 main 线程，但mwait被锁定，因此，main函数中的cnd_wait无法锁定mwait，阻塞线程，1秒后，mwait解锁，main函数继续执行。

线程局域存储

线程局域存储，可以在一个线程内存储内容。在线程终止时，会调用指定的函数，以便进行释放资源等操作（这个函数叫析构函数）。

存储键

线程需要有一个存储键才能存储。存储键的类型是**tss_t**。

存储键的创建

创建一个存储键采用**tss_create(3)**函数：

```
int tss_create( tss_t* tss_key, tss_dtor_t destructor );
```

其中，tss_t指向要创建的存储键的地址，destructor是线程结束后要调用的析构函数。注意，如果进程调用了exit(3)或在线程推出前就调用了tss_delete(3)函数，则不会执行析构函数。
析构函数的类型是void (*)(void *)，接受一个void *类型的参数（空指针，也就是存储的内容），没有返回值。

存储键的使用

我们可以写入存储键的内容，或读取存储键的内容。

存储键的写入

存储键的写入采用**tss_set(3)**函数：

```
int tss_set( tss_t tss_id, void *val );
```

它将会把tss_id存储键的值设置为val. 注意：不同线程可以使用同一个存储键，但是可以设置为不同的值。

存储键的读取

存储键的读取采用**tss_get(3)**函数：

```
void *tss_get( tss_t tss_key );
```

它将会返回tss_id存储键的值。

存储键的销毁

存储键的销毁采用tss_delete(3)函数：

```
void tss_delete( tss_t tss_id );
```

它将会销毁tss_id存储键。注意，销毁时不调用析构函数，因此，需要确保调用这个函数前，所有使用这个存储键的线程结束。

举个例子

我们来看以下代码：

```c
#include <threads.h>
#include <stdio.h>
#include <unistd.h>

tss_t key;

void des(void *con){
    printf("调用析构函数，内容：%d\n", (int)con);
}

int th1(void *args){
    tss_set(key, (void *)1);
    int con = (int)tss_get(key);
    printf("th1线程调用thrd_get函数，内容：%d\n", con);
    return 0;
}

int th2(void *args){
    tss_set(key, (void *)2);
    int con = (int)tss_get(key);
    printf("th2线程调用thrd_get函数，内容：%d\n", con);
    return 0;
}

int main(void){
    thrd_t t1, t2;
    tss_create(&key, des);
    thrd_create(&t1, th1, NULL);
    thrd_create(&t2, th2, NULL);
    thrd_join(t1, NULL);
    thrd_join(t2, NULL);
    tss_delete(key);
    return 0;
}

```

以下是代码的流程：

1. 定义一个存储键key。
2. 定义一个析构函数des。

3. 定义两个线程函数th1和th2。
    - 在th1中，线程将键key的值设置为1。然后它获取该键的值（应为1），将其强制转换为整型并打印出来。
    - 在th2中，线程将键key的值设置为2。然后它获取该键的值（应为2），将其强制转换为整型并打印出来。
4. 在main函数中：
    - 创建两个线程t1和t2，并分别将它们与线程函数th1和th2关联起来。
    - 对两个线程进行加入操作。
    - 使用tss_delete销毁存储键。

运行结果：

```
th2线程调用thrd_get函数，内容：2
调用析构函数，内容：2
th1线程调用thrd_get函数，内容：1
调用析构函数，内容：1
```

可以看到，当线程结束时调用析构函数，并且不同的线程使用相同的存储键时，可以存储不同的结果。