

第五章 线程编程

本章将分为两大部分进行讲解，前半部分将引出线程的使用场景及基本概念，通过示例代码来说明一个线程创建到退出到回收的基本流程。后半部分则会通过示例代码来说明如果控制好线程，从临界资源访问与线程的执行顺序控制上引出互斥锁、信号量的概念与使用方法。

5.1 线程的使用

5.1.1 为什么要使用多线程

在编写代码时，是否会遇到以下的场景会感觉到难以下手？

场景一：写程序在拷贝文件时，需要一边去拷贝文件，一边去向用户展示拷贝文件的进度时，传统做法是通过每次拷贝完成结束后去更新变量，再将变量转化为进度显示出来。其中经历了拷贝->计算->显示->拷贝->计算->显示...直至拷贝结束。这样的程序架构及其的低效，必须在单次拷贝结束后才可以刷新当前拷贝进度，若可以将进程分支，一支单独的解决拷贝问题，一支单独的解决计算刷新问题，则程序效率会提升很多。

场景二：用阻塞方式去读取数据，实时需要发送数据的时候。例如在进行串口数据传输或者网络数据传输的时候，我们往往需要双向通信，当设置读取数据为阻塞模式时候，传统的单线程只能等到数据接收来临后才能冲过阻塞，再根据逻辑进行发送。当我们要实现随时发送、随时接收时，无法满足我们的业务需求。若可以将进程分支，一支单纯的处理接收数据逻辑，一支单纯的解决发送数据逻辑，就可以完美的实现功能。

基于以上场景描述，多线程编程可以完美的解决上述问题。

5.1.2 线程概念

所谓线程，就是操作系统所能调度的最小单位。普通的进程，只有一个线程在执行对应的逻辑。我们可以通过多线程编程，使一个进程可以去执行多个不同的任务。相比多进程编程而言，线程享有共享资源，即在进程中出现的全局变量，每个线程都可以去访问它，与进程共享“4G”内存空间，使得系统资源消耗减少。本章节来讨论 Linux 下 POSIX 线程。

5.1.3 线程的标识 pthread_t

对于进程而言，每一个进程都有一个唯一对应的 PID 号来表示该进程，而对于线程而言，也有一个“类似于进程的 PID 号”，名为 tid，其本质是一个 pthread_t 类型的变量。线程号与进程号是表示线程和进程的唯一标识，但是对于线程号而言，其仅仅在其所属的进程上下文中才有意义。

获取线程号

```
#include <pthread.h>
```

```
pthread_t pthread_self(void);
```

成功：返回线程号

在程序中，可以通过函数，pthread_self，来返回当前线程的线程号，例程 1 给出了打印线程 tid 号。

测试例程 1：（Phtread_txex1.c）

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  int main()
5  {
6      pthread_t tid = pthread_self();//获取主线程的 tid 号
7      printf("tid = %lu\n",(unsigned long)tid);
8      return 0;
9  }
```

注意：因采用 POSIX 线程接口，故在要编译的时候包含 pthread 库，使用 gcc 编译应 gcc xxx.c -lpthread 方可编译多线程程序。

编译结果：

```
hbw@hbw-linux:Pthread$ gcc Pthread_Text1.c -lpthread
hbw@hbw-linux:Pthread$ ./a.out
tid = 139765901956864
```

5.1.4 线程的创建

创建线程

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *thread, const pthread_attr_t  
*attr, void *(*start_routine) (void *), void *arg);
```

成功：返回 0

在传统的程序中，一个进程只有一个线程，可以通过函数 `pthread_create` 来创建线程。

该函数第一个参数为 `pthread_t` 类型的线程号地址，当函数执行成功后会指向新建线程的线程号；第二个参数表示了线程的属性，一般传入 `NULL` 表示默认属性；第三个参数代表返回值为 `void*`，形参为 `void*` 的函数指针，当线程创建成功后，会自动的执行该回调函数；第四个参数则表示为向线程处理函数传入的参数，若不传入，可用 `NULL` 填充，有关线程传参后续小节会有详细的说明，接下来通过一个简单例程来使用该函数创建一个线程。

测试例程 2：（Phtread_txex2.c）

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <errno.h>
5
6  void *fun(void *arg)
7  {
8      printf("pthread_New = %lu\n", (unsigned long)pthread_self()); //打印线程的 tid 号
9  }
10
11 int main()
12 {
13
14     pthread_t tid1;
15     int ret = pthread_create(&tid1, NULL, fun, NULL); //创建线程
16     if (ret != 0) {
17         perror("pthread_create");
18         return -1;
19     }
20
```

```

21      /*tid_main 为通过 pthread_self 获取的线程 ID，tid_new 通过执行 pthread_create
    成功后 tid 指向的空间*/
22      printf("tid_main = %lu tid_new = %lu \n", (unsigned long)pthread_self(), (unsigned
    long)tid1);
23
24      /*因线程执行顺序随机，不加 sleep 可能导致主线程先执行，导致进程结束，无法
    执行到子线程*/
25      sleep(1);
26
27      return 0;
28  }
29

```

运行结果：

```

hbw@hbw-linux:Pthread$ ./a.out
tid_main = 139745244817152 tid_new = 139745236506368
pthread_New = 139745236506368

```

通过 `pthread_create` 确实可以创建出来线程，主线程中执行 `pthread_create` 后的 `tid` 指向了线程号空间，与子线程通过函数 `pthread_self` 打印出来的线程号一致。

特别说明的是，当主线程伴随进程结束时，所创建出来的线程也会立即结束，不会继续执行。并且创建出来的线程的执行顺序是随机竞争的，并不能保证哪一个线程会先运行。可以将上述代码中 `sleep` 函数进行注释，观察实验现象。

去掉上述代码 25 行后运行结果：

```

hbw@hbw-linux:Pthread$ ./a.out
tid_main = 139876185450240 tid_new = 139876177139456
hbw@hbw-linux:Pthread$ ./a.out
tid_main = 139940851799808 tid_new = 139940843489024
hbw@hbw-linux:Pthread$ ./a.out
tid_main = 139853357143808 tid_new = 139853348833024
pthread_New = 139853348833024
pthread_New = 139853348833024

```

上述运行代码 3 次，其中有 2 次被进程结束，无法执行到子线程的逻辑，最后一次则执行到了子线程逻辑后结束的进程。如此可以说明，线程的执行顺序不受控制，且整个进程结束后所产生的线程也随之被释放，在后续内容中将会描述如何控制线程执行。

5.1.5 向线程传入参数

`pthread_create()` 的最后一个参数的为 `void*` 类型的数据，表示可以向线程传递一个 `void*` 数据类型的参数，线程的回调函数中可以获取该参数，例程 3 举例了如何向线程传入变量地址与变量值。

测试例程 3: (Pthread_txex3.c)

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <errno.h>
5
6  void *fun1(void *arg)
7  {
8      printf("%s:arg = %d Addr = %p\n",__FUNCTION__,*(int *)arg,arg);
9  }
10
11 void *fun2(void *arg)
12 {
13     printf("%s:arg = %d Addr = %p\n",__FUNCTION__,(int)(long)arg,arg);
14 }
15
16 int main()
17 {
18
19     pthread_t tid1,tid2;
20     int a = 50;
21     int ret = pthread_create(&tid1,NULL,fun1,(void *)&a);//创建线程传入变量 a 的地址
22     if(ret != 0){
23         perror("pthread_create");
24         return -1;
25     }
27     ret = pthread_create(&tid2,NULL,fun2,(void *)a);//创建线程传入变量 a 的值
28     if(ret != 0){
29         perror("pthread_create");
30         return -1;
31     }
32     sleep(1);
33     printf("%s:a = %d Add = %p \n",__FUNCTION__,a,&a);
34     return 0;
35 }
36
```

运行结果:

```
hbw@hbw-linux:Pthread$ ./a.out
fun1:arg = 50 Addr = 0x7ffc3fede3d0
fun2:arg = 50 Addr = 0x32
main:a = 50 Add = 0x7ffc3fede3d0
```

本例程展示了如何利用线程创建函数的第四个参数向线程传入数据，举例了如何以地址的方式传入值、以变量的方式传入值，例程代码的 21 行，是将变量 **a** 先行取地址后，再次强制类型转化为 **void*** 后传入线程，线程处理的回调函数中，先将万能指针 **void*** 转化为 **int***，再次取地址就可以获得该地址变量的值，其本质在于地址的传递。例程代码的 27 行，直接将 **int** 类型的变量强制转化为 **void*** 进行传递（针对不同位数机器，指针对其字数不同，需要 **int** 转化为 **long** 在转指针，否则可能会发生警告），在线程处理回调函数中，直接将 **void*** 数据转化为 **int** 类型即可，本质上是在传递变量 **a** 的值。

上述两种方法均可得到所要的值，但是要注意其本质，一个为地址传递，一个为值的传递。当变量发生改变时候，传递地址后，该地址所对应的变量也会发生改变，但传入变量值的时候，即使地址指针所指的变量发生变化，但传入的为变量值，不会受到指针的指向的影响，实际项目中切记两者之间的区别。具体说明见例程 4。

测试例程 4：（Pthread_txex4.c）

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <errno.h>
5
6  void *fun1(void *arg)
7  {
8      while(1){
9
10         printf("%s:arg = %d Addr = %p\n",__FUNCTION__,*(int *)arg,arg);
11         sleep(1);
12     }
13 }
14
15 void *fun2(void *arg)
16 {
17     while(1){
18
19         printf("%s:arg = %d Addr = %p\n",__FUNCTION__,(int)(long)arg,arg);
20         sleep(1);
21     }
22 }
23
24 int main()
25 {
26
27     pthread_t tid1,tid2;
28     int a = 50;
29     int ret = pthread_create(&tid1,NULL,fun1,(void *)&a);
30     if(ret != 0){
31         perror("pthread_create");
```

```

32     return -1;
33 }
34 sleep(1);
35 ret = pthread_create(&tid2,NULL,fun2,(void *) (long)a);
36 if(ret != 0){
37     perror("pthread_create");
38     return -1;
39 }
40 while(1){
41     a++;
42     sleep(1);
43     printf("%s:a = %d Add = %p \n",__FUNCTION__,a,&a);
44 }
45 return 0;
46 }
47

```

运行结果:

```

hbw@hbw-linux:~$ ./a.out
fun1:arg = 50 Addr = 0x7fff0378ff40
fun1:arg = 50 Addr = 0x7fff0378ff40
fun2:arg = 50 Addr = 0x32
fun1:arg = 51 Addr = 0x7fff0378ff40
fun2:arg = 50 Addr = 0x32
main:a = 51 Add = 0x7fff0378ff40
fun1:arg = 52 Addr = 0x7fff0378ff40
main:a = 52 Add = 0x7fff0378ff40
fun2:arg = 50 Addr = 0x32
fun1:arg = 53 Addr = 0x7fff0378ff40
main:a = 53 Add = 0x7fff0378ff40
fun2:arg = 50 Addr = 0x32
fun1:arg = 54 Addr = 0x7fff0378ff40
fun2:arg = 50 Addr = 0x32
main:a = 54 Add = 0x7fff0378ff40
fun1:arg = 55 Addr = 0x7fff0378ff40
main:a = 55 Add = 0x7fff0378ff40
fun2:arg = 50 Addr = 0x32
fun1:arg = 56 Addr = 0x7fff0378ff40
fun2:arg = 50 Addr = 0x32
main:a = 56 Add = 0x7fff0378ff40
fun1:arg = 57 Addr = 0x7fff0378ff40
main:a = 57 Add = 0x7fff0378ff40
fun2:arg = 50 Addr = 0x32
^C

```

上述例程讲述了如何向线程传递一个参数, 在处理实际项目中, 往往会遇到传递多个参数的问题, 我们可以通过结构体来进行传递, 解决此问题。

测试例程 5: (Phtread_txex5.c)

```

1  #include <pthread.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <errno.h>
6
7  struct Stu{
8      int Id;
9      char Name[32];
10     float Mark;
11 };
12
13 void *fun1(void *arg)
14 {
15     struct Stu *tmp = (struct Stu *)arg;
16     printf("%s:Id      =      %d      Name      =      %s      Mark
= %.2f\n",__FUNCTION__,tmp->Id,tmp->Name,tmp->Mark);
17
18 }
19
20 int main()
21 {
22
23     pthread_t tid1,tid2;
24     struct Stu stu;
25     stu.Id = 10000;
26     strcpy(stu.Name,"ZhangSan");
27     stu.Mark = 94.6;
28
29     int ret = pthread_create(&tid1,NULL,fun1,(void *)&stu);
30     if(ret != 0){
31         perror("pthread_create");
32         return -1;
33     }
34     printf("%s:Id      =      %d      Name      =      %s      Mark
= %.2f\n",__FUNCTION__,stu.Id,stu.Name,stu.Mark);
35     sleep(1);
36     return 0;
37 }
38

```

运行结果:


```
hbw@hbw-linux:~$ ./a.out
main:Id = 10000 Name = ZhangSan Mark = 94.60
fun1:Id = 10000 Name = ZhangSan Mark = 94.60
```

5.1.6 线程的退出与回收

线程的退出情况有三种：第一种是进程结束，进程中所有的线程也会随之结束。第二种是通过函数 `pthread_exit` 来主动的退出线程。第三种通过函数 `pthread_cancel` 被其他线程被动结束。当线程结束后，主线程可以通过函数 `pthread_join`/`pthread_tryjoin_np` 来回收线程的资源，并且获得线程结束后需要返回的数据。

线程退出

```
#include <pthread.h>

void pthread_exit(void *retval);
```

该函数为线程退出函数，在退出时候可以传递一个 `void*` 类型的数据带给主线程，若选择不传出数据，可将参数填充为 `NULL`。

线程资源回收（阻塞）

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

成功：返回 0

该函数为线程回收函数，默认状态为阻塞状态，直到成功回收线程后被冲开阻塞。第一个参数为要回收线程的 `tid` 号，第二个参数为线程回收后接受线程传出的数据。

线程资源回收（非阻塞）

```
#define _GNU_SOURCE

#include <pthread.h>

int pthread_tryjoin_np(pthread_t thread, void **retval);
```

成功：返回 0

该函数为非阻塞模式回收函数，通过返回值判断是否回收掉线程，成功回收则返回 0，其余参数与 `pthread_join` 一致。

线程退出（指定线程号）

```
#include <pthread.h>
```

```
int pthread_cancel(pthread_t thread);
```

成功：返回 0

该函数传入一个 tid 号，会强制退出该 tid 所指向的线程，若成功执行会返回 0。

上述描述简单的介绍了有关线程回收的 API，下面通过例程来说明上述 API。

测试例程 6：（Phtread_txex6.c）

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <errno.h>
5
6  void *fun1(void *arg)
7  {
8      static int tmp = 0; //必须要 static 修饰，否则 pthread_join 无法获取到正确值
9      //int tmp = 0;
10     tmp = *(int *)arg;
11     tmp += 100;
12     printf("%s:Addr = %p tmp = %d\n", __FUNCTION__, &tmp, tmp);
13     pthread_exit((void *)&tmp); //将变量 tmp 取地址转化为 void* 类型传出
14 }
15
16
17 int main()
18 {
19
20     pthread_t tid1;
21     int a = 50;
22     void *Tmp = NULL; //因 pthread_join 第二个参数为 void** 类型
23     int ret = pthread_create(&tid1, NULL, fun1, (void *)&a);
24     if (ret != 0) {
25         perror("pthread_create");
26         return -1;
27     }
28     pthread_join(tid1, &Tmp);
29     printf("%s:Addr = %p Val = %d\n", __FUNCTION__, Tmp, *(int *)Tmp);
30     return 0;
31 }
```

运行结果：

```
hbw@hbw-linux:~$ ./a.out
fun1:Addr = 0x601064 tmp = 150
main:Addr = 0x601064 Val = 150
```

上述例程先通过 23 行将变量以地址的形式传入线程，在线程中做出了自加 100 的操作，当线程退出的时候通过线程传参，用 `void*` 类型的数据通过 `pthread_join` 接受。此例程去掉了之前加入的 `sleep` 函数，原因是 `pthread_join` 函数具备阻塞的特性，直至成功收回掉线程后才会冲破阻塞，因此不需要考虑主线程会执行到 30 行结束进程的情况。特别要说明的是例程第 8 行，当变量从线程传出的时候，需要加 `static` 修饰，对生命周期做出延续，否则无法传出正确的变量值。

测试例程 7：（Pthread_txex7.c）

```
1  #define _GNU_SOURCE
2  #include <pthread.h>
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <errno.h>
6
7  void *fun(void *arg)
8  {
9      printf("Pthread:%d Come !\n",(int )(long)arg+1);
10     pthread_exit(arg);
11 }
12
13
14 int main()
15 {
16     int ret,i,flag = 0;
17     void *Tmp = NULL;
18     pthread_t tid[3];
19     for(i = 0;i < 3;i++){
20         ret = pthread_create(&tid[i],NULL,fun,(void *)(long)i);
21         if(ret != 0){
22             perror("pthread_create");
23             return -1;
24         }
25     }
26     while(1){//通过非阻塞方式收回线程，每次成功回收一个线程变量自增，直至 3 个
线程全数回收
27         for(i = 0;i < 3;i++){
28             if(pthread_tryjoin_np(tid[i],&Tmp) == 0){
```

```

29             printf("Pthread : %d exit !\n",(int )(long )Tmp+1);
30             flag++;
31         }
32     }
33     if(flag >= 3) break;
34 }
35 return 0;
36 }
37

```

运行结果：

```

hbw@hbw-linux:Pthread$ ./a.out
Pthread:1 Come !
Pthread : 1 exit !
Pthread:2 Come !
Pthread:3 Come !
Pthread : 2 exit !
Pthread : 3 exit !

```

例程 7 展示了如何使用非阻塞方式来回收线程,此外也展示了多个线程可以指向同一个回调函数的情况。例程 6 通过阻塞方式回收线程几乎规定了线程回收的顺序,若最先回收的线程未退出,则一直会被阻塞,导致后续先退出的线程无法及时的回收。

通过函数 `pthread_tryjoin_np`,使用非阻塞回收,线程可以根据退出先后顺序自由的进行资源的回收。

测试例程 8: (Phtread_txex8.c)

```

1  #define _GNU_SOURCE
2  #include <pthread.h>
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <errno.h>
6
7  void *fun1(void *arg)
8  {
9      printf("Pthread:1 come!\n");
10     while(1){
11         sleep(1);
12     }
13 }
14
15 void *fun2(void *arg)
16 {
17     printf("Pthread:2 come!\n");
18     pthread_cancel((pthread_t )(long)arg);//杀死线程 1, 使之强制退出
19     pthread_exit(NULL);

```

```

20 }
21
22 int main()
23 {
24     int ret,i,flag = 0;
25     void *Tmp = NULL;
26     pthread_t tid[2];
27     ret = pthread_create(&tid[0],NULL,fun1,NULL);
28     if(ret != 0){
29         perror("pthread_create");
30         return -1;
31     }
32     sleep(1);
33     ret = pthread_create(&tid[1],NULL,fun2,(void *)tid[0]);//传输线程 1 的线程号
34     if(ret != 0){
35         perror("pthread_create");
36         return -1;
37     }
38     while(1){//通过非阻塞方式收回线程，每次成功回收一个线程变量自增，直至 2 个
线程全数回收
39         for(i = 0;i < 2;i++){
40             if(pthread_tryjoin_np(tid[i],NULL) == 0){
41                 printf("Pthread : %d exit !\n",i+1);
42                 flag++;
43             }
44         }
45         if(flag >= 2) break;
46     }
47     return 0;
48 }
49

```

运行结果：

```

hbw@hbw-linux:~$ ./a.out
Pthread:1 come!
Pthread:2 come!
Pthread : 1 exit !
Pthread : 2 exit !

```

例程 8 展示了如何利用 `pthread_cancel` 函数主动的将某个线程结束。27 行与 33 行创建了线程，将第一个线程的线程号传参形式传入了第二个线程。第一个的线程执行死循环睡眠逻辑，理论上除非进程结束，其永远不会结束，但在第二个线程中调用了 `pthread_cancel` 函数，相当于向该线程发送一个退出的指令，导致线程被退出，最终资源被非阻塞回收掉。此例程要注意第 32 行的 `sleep` 函数，一定要确保线程 1 先执行，因线程是无序执行，故加入

该睡眠函数控制顺序，在本章后续，会讲解通过加锁、信号量等手段来合理的控制线程的临界资源访问与线程执行顺序控制。

5.2 线程的控制

5.2.1 多线程编临界资源访问

当线程在运行过程中，去操作公共资源，如全局变量的时候，可能会发生彼此“矛盾”现象。例如线程 1 企图想让变量自增，而线程 2 企图想要变量自减，两个线程存在互相竞争的关系导致变量永远处于一个“平衡状态”，两个线程互相竞争，线程 1 得到执行权后将变量自加，当线程 2 得到执行权后将变量自减，变量似乎永远在某个范围内浮动，无法到达期望数值，如例程 9 所示。

测试例程 9：（Pthread_txex9.c）

```
1  #define _GNU_SOURCE
2  #include <pthread.h>
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <errno.h>
6
7
8  int Num = 0;
9
10 void *fun1(void *arg)
11 {
12     while(Num < 3){
13         Num++;
14         printf("%s:Num = %d\n",__FUNCTION__,Num);
15         sleep(1);
16     }
17     pthread_exit(NULL);
18 }
19
20 void *fun2(void *arg)
21 {
22     while(Num > -3){
23         Num--;
24         printf("%s:Num = %d\n",__FUNCTION__,Num);
25         sleep(1);
26     }
27     pthread_exit(NULL);
```

```

28 }
29
30 int main()
31 {
32     int ret;
33     pthread_t tid1,tid2;
34     ret = pthread_create(&tid1,NULL,fun1,NULL);
35     if(ret != 0){
36         perror("pthread_create");
37         return -1;
38     }
39     ret = pthread_create(&tid2,NULL,fun2,NULL);
40     if(ret != 0){
41         perror("pthread_create");
42         return -1;
43     }
44     pthread_join(tid1,NULL);
45     pthread_join(tid2,NULL);
46     return 0;
47 }
48

```

运行结果：

```

hbw@hbw-linux:~$ ./a.out
fun2:Num = -1
fun1:Num = 1
fun2:Num = 0
fun1:Num = 1
fun2:Num = 0
fun1:Num = 1
fun1:Num = 2
fun2:Num = 1
fun1:Num = 2
fun2:Num = 1
fun2:Num = 0
fun1:Num = 1
fun2:Num = 0
fun1:Num = 1
^C

```

为了解决上述对临界资源的竞争问题，pthread 线程引出了互斥锁来解决临界资源访问。通过对临界资源加锁来保护资源只被单个线程操作，待操作结束后解锁，其余线程才可获得操作权。

5.2.2 互斥锁 API 简述

初始化互斥锁

```
#include <pthread.h>
```

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
                        const pthread_mutexattr_t *restrict attr);
```

成功：返回 0

该函数作用为初始化一个互斥锁，一般情况申请一个全局的 `pthread_mutex_t` 类型的互斥锁变量，通过此函数完成锁内的初始化，第一个函数将该变量的地址传入，第二个参数为控制互斥锁的属性，一般为 `NULL`。当函数成功后会返回 0，代表初始化互斥锁成功。当然初始化互斥锁也可以调用宏来快速初始化：

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

互斥锁加锁（阻塞）/解锁

```
#include <pthread.h>
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

`lock` 函数与 `unlock` 函数分别为加锁解锁函数，只需要传入已经初始化好的 `pthread_mutex_t` 互斥锁变量，成功后会返回 0。当某一个线程获得了执行权后，执行 `lock` 函数，一旦加锁成功后，其余线程遇到 `lock` 函数时候会发生阻塞，直至获取资源的线程执行 `unlock` 函数后，获得第二执行权的线程的阻塞模式被从开，同时也获取了 `lock`，导致其余线程同样在阻塞，直至执行 `unlock` 被解锁。

特别注意的是，当获取 `lock` 之后，必须在逻辑处理结束后执行 `unlock`，否则会发生死锁现象！导致其余线程一直处于阻塞状态，无法执行下去。在使用互斥锁的时候，尤其要注意使用 `pthread_cancel` 函数，防止发生死锁现象！

互斥锁加锁（非阻塞）

```
#include <pthread.h>
```

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

成功：返回 0

该函数同样也是一个线程加锁函数，但该函数是非阻塞模式通过返回值来判断是否加锁成功，用法与上述阻塞加速函数一致。

互斥锁销毁

```
#include <pthread.h>
```

```
int pthread_mutex_destory(pthread_mutex_t *mutex);
```

成功：返回 0

该函数是用于销毁互斥锁的，传入互斥锁的地址，就可以完成互斥锁的销毁，成功返回 0。

测试例程 10：（Phtread_txex10.c）

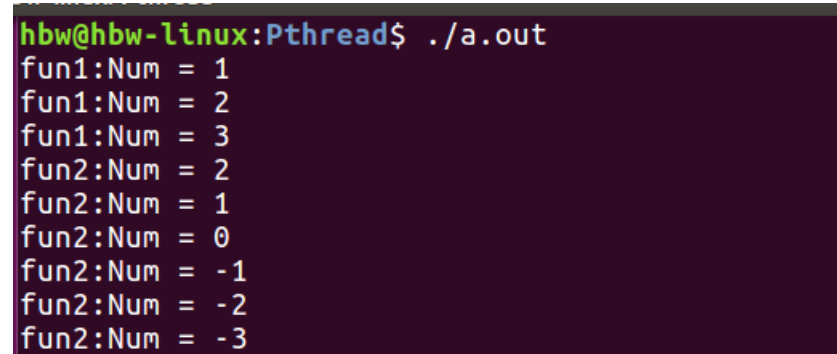
```
1  #define _GNU_SOURCE
2  #include <pthread.h>
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <errno.h>
6
7  pthread_mutex_t mutex;//互斥锁变量 一般申请全局变量
8
9  int Num = 0;//公共临界变量
10
11 void *fun1(void *arg)
12 {
13     pthread_mutex_lock(&mutex);//加锁 若有线程获得锁，则会阻塞
14     while(Num < 3){
15         Num++;
16         printf("%s:Num = %d\n",__FUNCTION__,Num);
17         sleep(1);
18     }
19     pthread_mutex_unlock(&mutex);//解锁
20     pthread_exit(NULL);//线程退出 pthread_join 会回收资源
21 }
22
23 void *fun2(void *arg)
24 {
25     pthread_mutex_lock(&mutex);//加锁 若有线程获得锁，则会阻塞
26     while(Num > -3){
27         Num--;
28         printf("%s:Num = %d\n",__FUNCTION__,Num);
29         sleep(1);
30     }
31     pthread_mutex_unlock(&mutex);//解锁
```

```

32     pthread_exit(NULL);//线程退出 pthread_join 会回收资源
33 }
34
35 int main()
36 {
37     int ret;
38     pthread_t tid1,tid2;
39     ret = pthread_mutex_init(&mutex,NULL);//初始化互斥锁
40     if(ret != 0){
41         perror("pthread_mutex_init");
42         return -1;
43     }
44     ret = pthread_create(&tid1,NULL,fun1,NULL);//创建线程 1
45     if(ret != 0){
46         perror("pthread_create");
47         return -1;
48     }
49     ret = pthread_create(&tid2,NULL,fun2,NULL);//创建线程 2
50     if(ret != 0){
51         perror("pthread_create");
52         return -1;
53     }
54     pthread_join(tid1,NULL);//阻塞回收线程 1
55     pthread_join(tid2,NULL);//阻塞回收线程 2
56     pthread_mutex_destroy(&mutex);//销毁互斥锁
57     return 0;
58 }
59

```

运行结果:



```

hbw@hbw-linux:~$ ./a.out
fun1:Num = 1
fun1:Num = 2
fun1:Num = 3
fun2:Num = 2
fun2:Num = 1
fun2:Num = 0
fun2:Num = -1
fun2:Num = -2
fun2:Num = -3

```

上述例程通过加入互斥锁，保证了临界变量某时刻只被某一线程控制，实现了临界资源的控制。需要说明的是，线程加锁在循环内与循环外的情况。本历程在进入 while 循环前进行了加锁操作，在循环结束后进行的解锁操作，如果将加锁解锁全部放入 while 循环内，作为单核的机器，执行结果无异，当有多核机器执行代码时，可能会发生“抢锁”现象，这取决于操作系统底层的实现。

5.2.3 多线程编执行顺序控制

解决了临界资源的访问，但似乎对线程的执行顺序无法得到控制，因线程都是无序执行，之前采用 `sleep` 强行延时的方法勉强可以控制执行顺序，但此方法在实际项目情况往往是不可取的，其仅仅可解决线程创建的顺序，当创建之后执行的顺序又不会受到控制，于是便引入了信号量的概念，解决线程执行顺序。

例程 11 将展示线程的执行的随机性。

测试例程 11：（Pthread_txex11.c）

```
1  #define _GNU_SOURCE
2  #include <pthread.h>
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <errno.h>
6
7  void *fun1(void *arg)
8  {
9      printf("%s:Pthread Come!\n",__FUNCTION__);
10     pthread_exit(NULL);
11 }
12
13 void *fun2(void *arg)
14 {
15     printf("%s:Pthread Come!\n",__FUNCTION__);
16     pthread_exit(NULL);
17 }
18
19 void *fun3(void *arg)
20 {
21     printf("%s:Pthread Come!\n",__FUNCTION__);
22     pthread_exit(NULL);
23 }
24
25 int main()
26 {
27     int ret;
28     pthread_t tid1,tid2,tid3;
29     ret = pthread_create(&tid1,NULL,fun1,NULL);
30     if(ret != 0){
31         perror("pthread_create");
32         return -1;
```

```

33     }
34     ret = pthread_create(&tid2,NULL,fun2,NULL);
35     if(ret != 0){
36         perror("pthread_create");
37         return -1;
38     }
39     ret = pthread_create(&tid3,NULL,fun3,NULL);
40     if(ret != 0){
41         perror("pthread_create");
42         return -1;
43     }
44     pthread_join(tid1,NULL);
45     pthread_join(tid2,NULL);
46     pthread_join(tid3,NULL);
47     return 0;
48 }
49

```

运行结果：

```

hbw@hbw-linux:~$ ./a.out
fun1:Pthread Come!
fun2:Pthread Come!
fun3:Pthread Come!
hbw@hbw-linux:~$ ./a.out
fun1:Pthread Come!
fun3:Pthread Come!
fun2:Pthread Come!
hbw@hbw-linux:~$ ./a.out
fun2:Pthread Come!
fun3:Pthread Come!
fun1:Pthread Come!
hbw@hbw-linux:~$ ./a.out
fun1:Pthread Come!
fun3:Pthread Come!
fun2:Pthread Come!
hbw@hbw-linux:~$ ./a.out
fun1:Pthread Come!
fun2:Pthread Come!
fun3:Pthread Come!

```

通过上述例程可以发现，多次执行该函数其次序是无序的，线程之间的竞争无法控制，通过使用信号量来使得线程顺序为可控的。

5.2.4.信号量 API 简述

初始化信号量

```
#include <semaphore.h>
```

```
int sem_init(sem_t *sem,int pshared,unsigned int value);
```

成功：返回 0

该函数可以初始化一个信号量，第一个参数传入 `sem_t` 类型的地址，第二个参数传入 0 代表线程控制，否则为进程控制，第三个参数表示信号量的初始值，0 代表阻塞，1 代表运行。待初始化结束信号量后，若执行成功会返回 0。

信号量 PV 操作（阻塞）

```
#include <pthread.h>
```

```
int sem_wait(sem_t *sem);
```

```
int sem_post(sem_t *sem);
```

成功：返回 0

`sem_wait` 函数作用为检测指定信号量是否有资源可用，若无资源可用会阻塞等待，若有资源可用会自动的执行“`sem-1`”的操作。所谓的“`sem-1`”是与上述初始化函数中第三个参数值一致，成功执行会返回 0。

`sem_post` 函数会释放指定信号量的资源，执行“`sem+1`”操作。

通过以上 2 个函数可以完成所谓的 PV 操作，即信号量的申请与释放，完成对线程执行顺序的控制。

信号量申请资源（非阻塞）

```
#include <pthread.h>
```

```
int sem_trywait(sem_t *sem);
```

成功：返回 0

与互斥锁一样，此函数是控制信号量申请资源的非阻塞函数，功能与 `sem_wait` 一致，唯一区别在于此函数为非阻塞。

信号量销毁

```
#include <pthread.h>
```

```
int sem_destory(sem_t *sem);
```

成功：返回 0

该函数为信号量销毁函数，执行过后可将申请的信号量进行销毁。

测试例程 12：（Phtread_txex12.c）

```
1  #define _GNU_SOURCE
2  #include <pthread.h>
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <errno.h>
6  #include <semaphore.h>
7
8  sem_t sem1,sem2,sem3;//申请的三个信号量变量
9
10 void *fun1(void *arg)
11 {
12     sem_wait(&sem1);//因 sem1 本身有资源，所以不被阻塞 获取后 sem1-1 下次会会
阻塞
13     printf("%s:Pthread Come!\n",__FUNCTION__);
14     sem_post(&sem2);// 使得 sem2 获取到资源
15     pthread_exit(NULL);
16 }
17
18 void *fun2(void *arg)
19 {
```

```

20     sem_wait(&sem2);//因 sem2 在初始化时无资源会被阻塞，直至 14 行代码执行 不
被阻塞 sem2-1 下次会阻塞
21     printf("%s:Pthread Come!\n",__FUNCTION__);
22     sem_post(&sem3);// 使得 sem3 获取到资源
23     pthread_exit(NULL);
24 }
25
26 void *fun3(void *arg)
27 {
28     sem_wait(&sem3);//因 sem3 在初始化时无资源会被阻塞，直至 22 行代码执行 不
被阻塞 sem3-1 下次会阻塞
29     printf("%s:Pthread Come!\n",__FUNCTION__);
30     sem_post(&sem1);// 使得 sem1 获取到资源
31     pthread_exit(NULL);
32 }
33
34 int main()
35 {
36     int ret;
37     pthread_t tid1,tid2,tid3;
38     ret = sem_init(&sem1,0,1); //初始化信号量 1 并且赋予其资源
39     if(ret < 0){
40         perror("sem_init");
41         return -1;
42     }
43     ret = sem_init(&sem2,0,0); //初始化信号量 2 让其阻塞
44     if(ret < 0){
45         perror("sem_init");
46         return -1;
47     }
48     ret = sem_init(&sem3,0,0); //初始化信号 3 让其阻塞
49     if(ret < 0){
50         perror("sem_init");
51         return -1;
52     }
53     ret = pthread_create(&tid1,NULL,fun1,NULL);//创建线程 1
54     if(ret != 0){
55         perror("pthread_create");
56         return -1;
57     }
58     ret = pthread_create(&tid2,NULL,fun2,NULL);//创建线程 2
59     if(ret != 0){
60         perror("pthread_create");
61         return -1;

```

```

62     }
63     ret = pthread_create(&tid3,NULL,fun3,NULL);//创建线程 3
64     if(ret != 0){
65         perror("pthread_create");
66         return -1;
67     }
68     /*回收线程资源*/
69     pthread_join(tid1,NULL);
70     pthread_join(tid2,NULL);
71     pthread_join(tid3,NULL);
72
73     /*销毁信号量*/
74     sem_destroy(&sem1);
75     sem_destroy(&sem2);
76     sem_destroy(&sem3);
77
78     return 0;
79 }
80

```

运行结果:

```

hbw@hbw-linux:~$ ./a.out
fun1:Pthread Come!
fun2:Pthread Come!
fun3:Pthread Come!
hbw@hbw-linux:~$ ./a.out
fun1:Pthread Come!
fun2:Pthread Come!
fun3:Pthread Come!
hbw@hbw-linux:~$ ./a.out
fun1:Pthread Come!
fun2:Pthread Come!
fun3:Pthread Come!
hbw@hbw-linux:~$ ./a.out
fun1:Pthread Come!
fun2:Pthread Come!
fun3:Pthread Come!
hbw@hbw-linux:~$ ./a.out
fun1:Pthread Come!
fun2:Pthread Come!
fun3:Pthread Come!

```

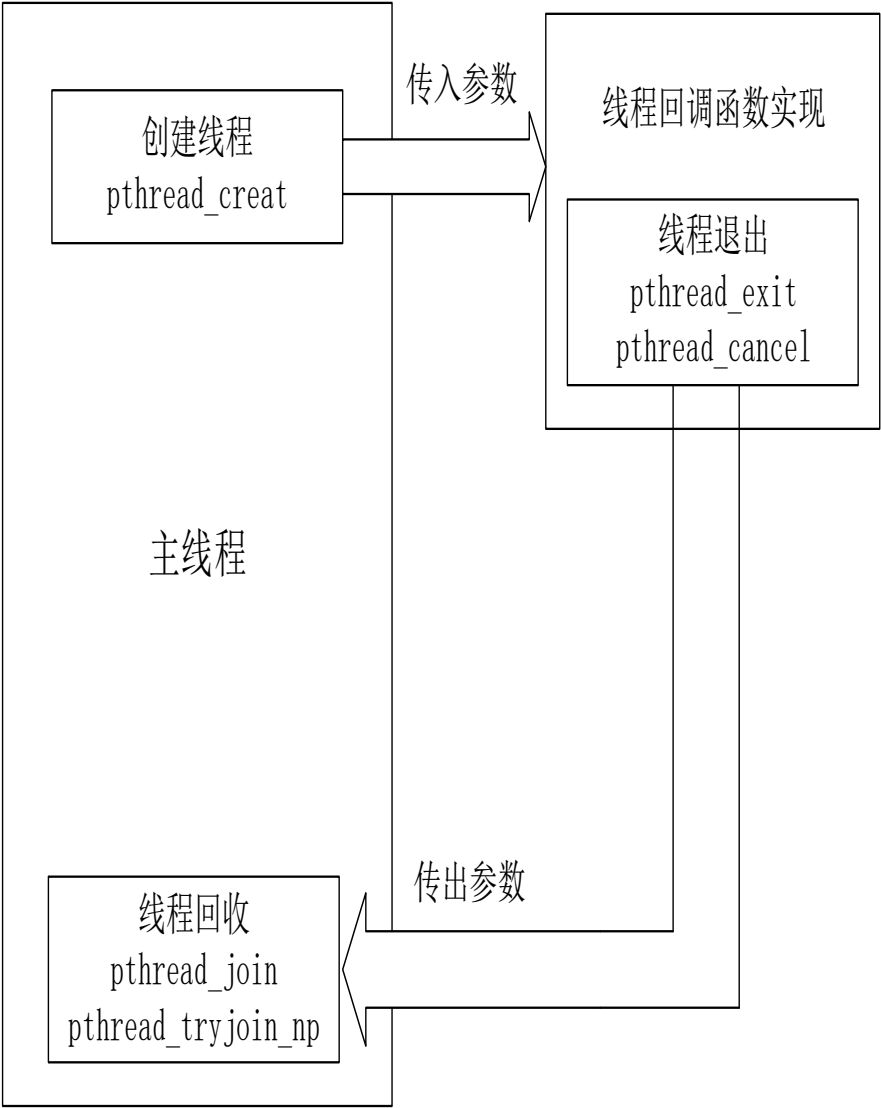
该例程加入了信号量的控制使得线程的执行顺序变为可控的，在初始化信号量时，将信号量 1 填入资源，使之不被 `sem_wait` 函数阻塞，在执行完逻辑后使用 `sem_post` 函数来填入即将要执行的资源。当执行函数 `sem_wait` 后，会执行 `sem` 自减操作，使下一次竞争被阻塞，直至通过 `sem_post` 被释放。

上述例程因 38 行初始化信号量 1 时候，使其默认获取到资源，43、48 行初始化信号量 2、3 时候，使之没有资源。于是在线程处理函数中，每个线程通过 `sem_wait` 函数来等待资

源，发送阻塞现象。因信号量 1 初始值为有资源，故可以先执行线程 1 的逻辑。待执行完第 12 行 `sem_wait` 函数，会导致 `sem1-1`，使得下一次此线程会被阻塞。继而执行至 14 行，通过 `sem_post` 函数使 `sem2` 信号量获取资源，从而冲破阻塞执行线程 2 的逻辑...以此类推完成线程的有序控制。

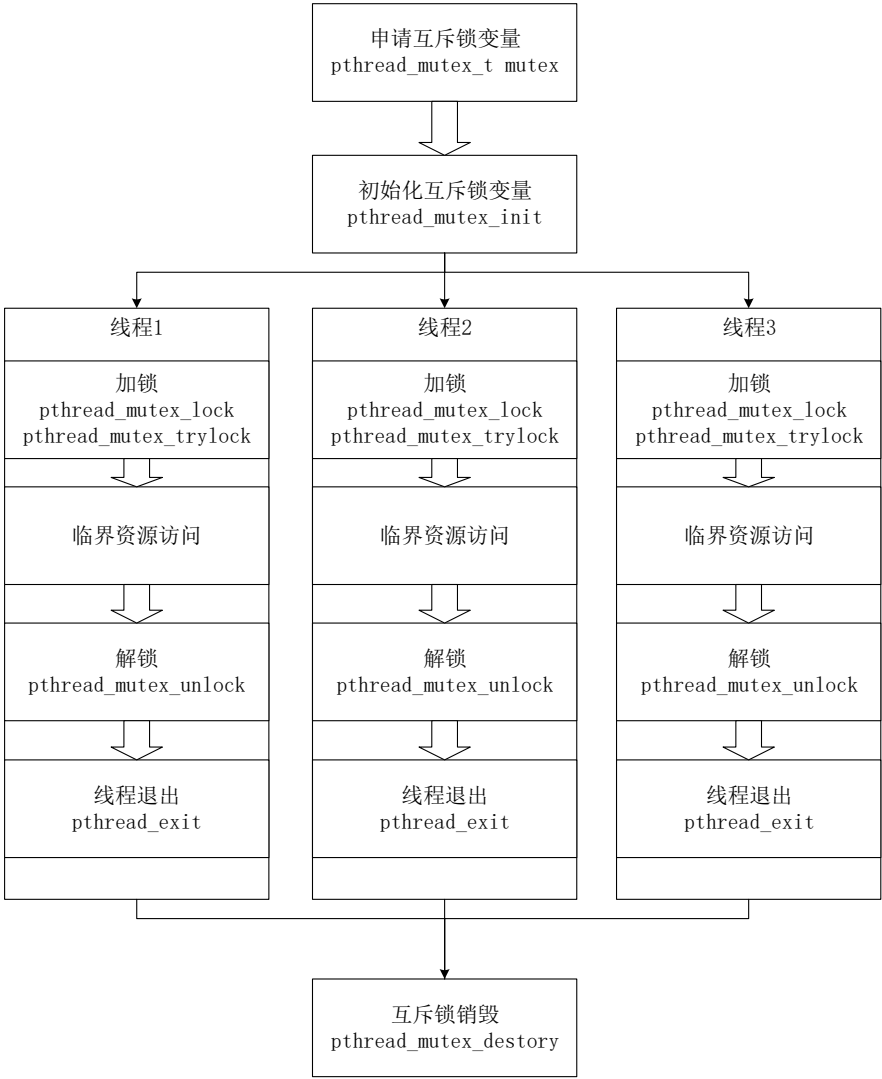
5.3 总结

有关多线程的创建流程下图所示，首先需要创建线程，一旦线程创建完成后，线程与线程之间会发生竞争执行，抢占时间片来执行线程逻辑。在创建线程时候，可以通过创建线程的第四个参数传入参数，在线程退出时亦可传出参数被线程回收函数所回收，获取到传出的参数。



线程编程流程

当多个线程出现后，会遇到同时操作临界公共资源的问题，当线程操作公共资源时需要对线程进行保护加锁，防止其与线程在此线程更改变量时同时更改变量，待逻辑执行完毕后再次解锁，使其余线程再度开始竞争。互斥锁创建流程下图所示。



互斥锁编程流程

当多个线程出现后，同时会遇到无序执行的问题。有时候需要对线程的执行顺序做出限定，变引入了信号量，通过 PV 操作来控制线程的执行顺序，下图所示。

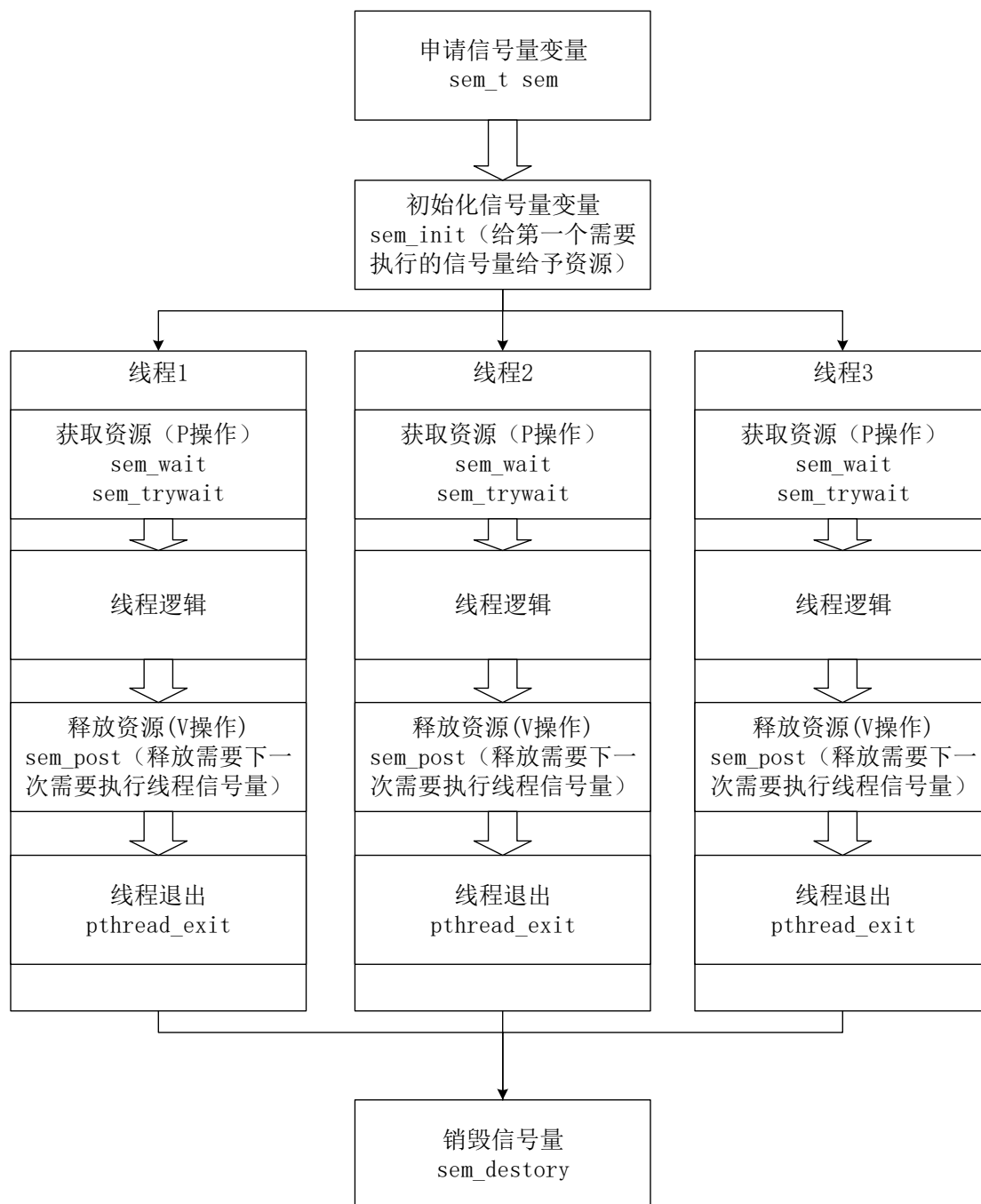


图 3 信号量编程流程