入内存，如果运行进程A后需要紧接着运行进程B，就应该将进程A相关信息移出内存，并读入进程B相关信息。这就是上下文切换。但此时进程A的数据将被移动到硬盘，所以上下文切换需要很长时间。即使通过优化加快速度，也会存在一定的局限。

上下文切换

通过学习计算机结构和操作系统相关知识，可以了解到上下文切换中具体的工作过程。但我为了讲述网络编程，只介绍了基础概念。实际上该过程应该通过 CPU 内部的寄存器来解释。

为了保持多进程的优点，同时在一定程度上克服其缺点，人们引入了线程（Thread）。这是为了将进程的各种劣势降至最低限度（不是直接消除）而设计的一种“轻量级进程”。线程相比于进程具有如下优点。

- 线程的创建和上下文切换比进程的创建和上下文切换更快。
- 线程间交换数据时无需特殊技术。

各位会逐渐体会到这些优点，可以通过接下来的说明和线程相关代码进行准确理解。

线程和进程的差异

线程是为了解决如下困惑登场的:

“嘿！为了得到多条代码执行流而复制整个内存区域的负担太重了！”

每个进程的内存空间都由保存全局变量的“数据区”、向malloc等函数的动态分配提供空间的堆（Heap）、函数运行时使用的栈（Stack）构成。每个进程都拥有这种独立空间，多个进程的内存结构如图18-1所示。

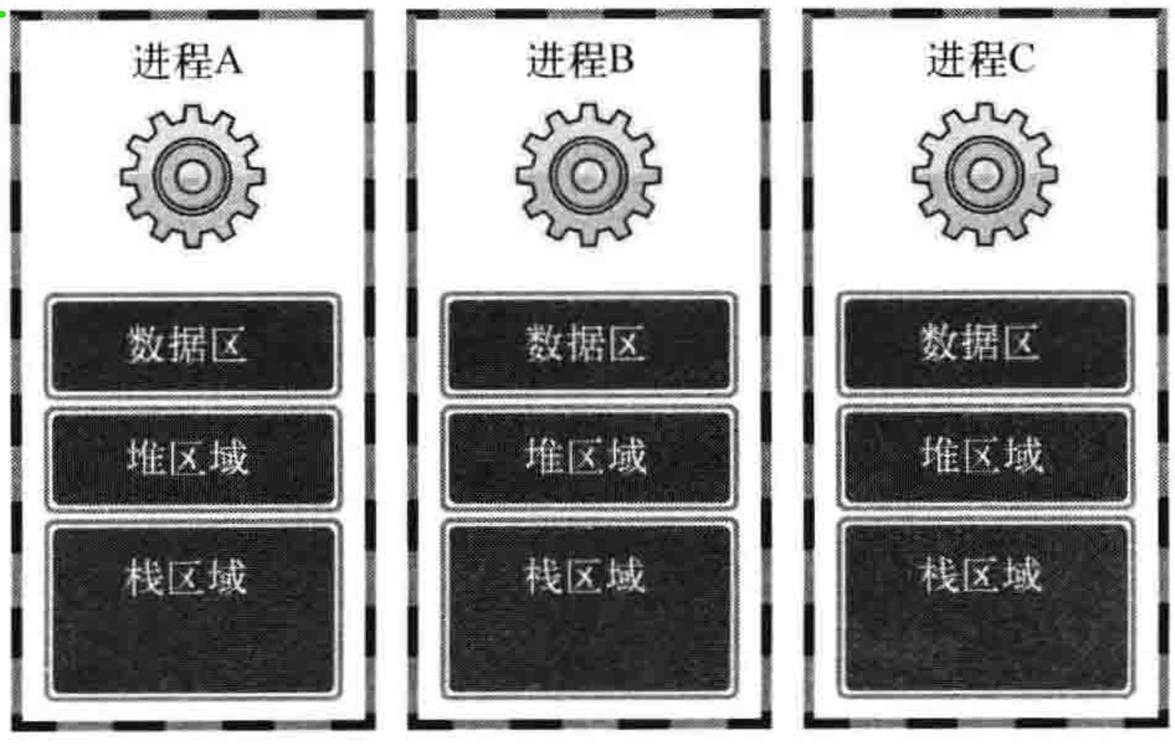

图18-1 进程间独立的内存

但如果以获得多个代码执行流为主要目的，则不应该像图18-1那样完全分离内存结构，而只需分离栈区域。通过这种方式可以获得如下优势。

- 上下文切换时不需要切换数据区和堆。
- 可以利用数据区和堆交换数据。

实际上这就是线程。线程为了保持多条代码执行流而隔开了栈区域，因此具有如图18-2所示的内存结构。

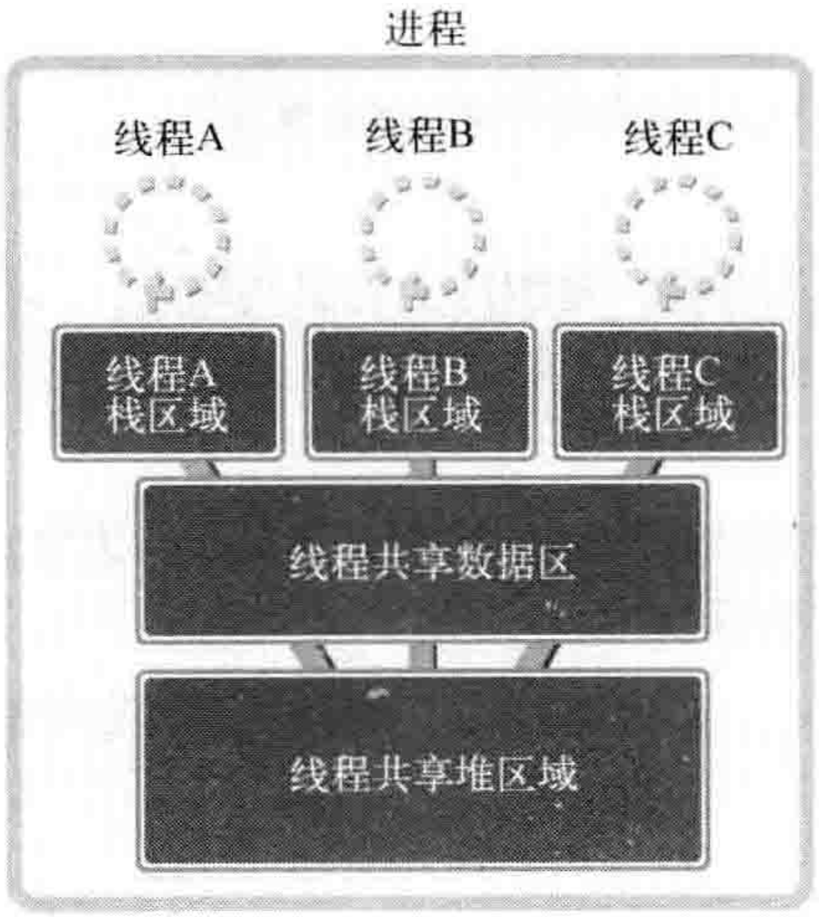

图18-2　线程的内存结构

如图18-2所示，多个线程将共享数据区和堆。为了保持这种结构，线程将在进程内创建并运行。也就是说，进程和线程可以定义为如下形式。

- 进程：在操作系统构成单独执行流的单位。
- 线程：在进程构成单独执行流的单位。

如果说进程在操作系统内部生成多个执行流，那么线程就在同一进程内部创建多条执行流。因此，操作系统、进程、线程之间的关系可以通过图18-3表示。

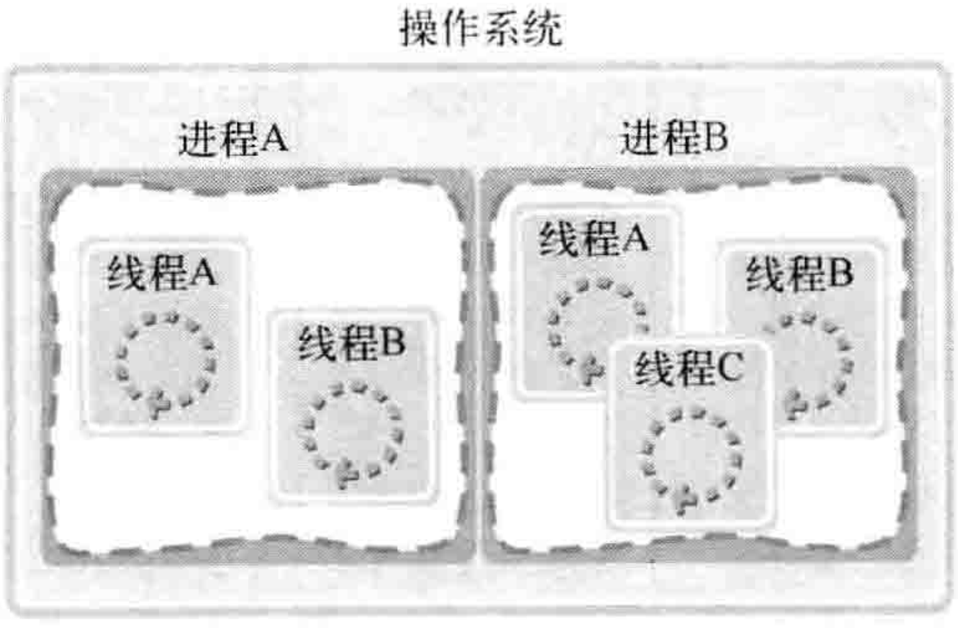

图18-3　操作系统、进程、线程之间的关系

以上就是线程的理论说明。没有实际编程就很难理解好线程，希望各位通过学习线程相关代码理解全部内容。

18.2 线程创建及运行

POSIX是Portable Operating System Interface for Computer Environment（适用于计算机环境的可移植操作系统接口）的简写，是为了提高UNIX系列操作系统间的移植性而制定的API规范。下面要介绍的线程创建方法也是以POSIX标准为依据的。因此，它不仅适用于Linux，也适用于大部分UNIX系列的操作系统。

线程的创建和执行流程

线程具有单独的执行流，因此需要单独定义线程的main函数，还需要请求操作系统在单独的执行流中执行该函数，完成该功能的函数如下。

```
#include <pthread.h>

int pthread_create(
    pthread_t * restrict thread, const pthread_attr_t * restrict attr,
    void * (* start_routine)(void *), void * restrict arg
);
    ➔ 成功时返回 0，失败时返回其他值。
```

- thread：保存新创建线程ID的变量地址值。线程与进程相同，也需要用于区分不同线程的ID。
- attr：用于传递线程属性的参数，传递NULL时，创建默认属性的线程。
- start_routine：相当于线程main函数的、在单独执行流中执行的函数地址值（函数指针）。
- arg：通过第三个参数传递调用函数时包含传递参数信息的变量地址值。

要想理解好上述函数的参数，需要熟练掌握restrict关键字和函数指针相关语法。但如果只关注使用方法（当然以后要掌握restrict和函数指针），那么该函数的使用比想象中要简单。下面通过简单示例了解该函数的功能。

❖ thread1.c

```
1.  #include <stdio.h>
2.  #include <pthread.h>
3.  void* thread_main(void *arg);
4.
5.  int main(int argc, char *argv[])
6.  {
```

```
7.      pthread_t t_id;
8.      int thread_param=5;
9.
10.     if(pthread_create(&t_id, NULL, thread_main, (void*)&thread_param)!=0)
11.     {
12.         puts("pthread_create() error");
13.         return -1;
14.     };
15.     sleep(10); puts("end of main");
16.     return 0;
17. }
18.
19. void* thread_main(void *arg)
20. {
21.     int i;
22.     int cnt=*((int*)arg);
23.     for(i=0; i<cnt; i++)
24.     {
25.         sleep(1); puts("running thread");
26.     }
27.     return NULL;
28. }
```

代码说明

- 第10行：请求创建一个线程，从thread_main函数调用开始，在单独的执行流中运行。同时在调用thread_main函数时向其传递thread_param变量的地址值。
- 第15行：调用sleep函数使main函数停顿10秒，这是为了延迟进程的终止时间。执行第16行的return语句后终止进程，同时终止内部创建的线程。因此，为保证线程的正常执行而添加这条语句。
- 第19、22行：传入arg参数的是第10行pthread_create函数的第四个参数。

❖ 运行结果：thread1.c

```
root@my_linux:/tcpip# gcc thread1.c -o tr1 -lpthread
root@my_linux:/tcpip# ./tr1
running thread
running thread
running thread
running thread
running thread
end of main
```

从上述运行结果中可以看到，线程相关代码在编译时需要添加–lpthread选项声明需要连接线程库，只有这样才能调用头文件pthread.h中声明的函数。上述程序的执行流程如图18-4所示。

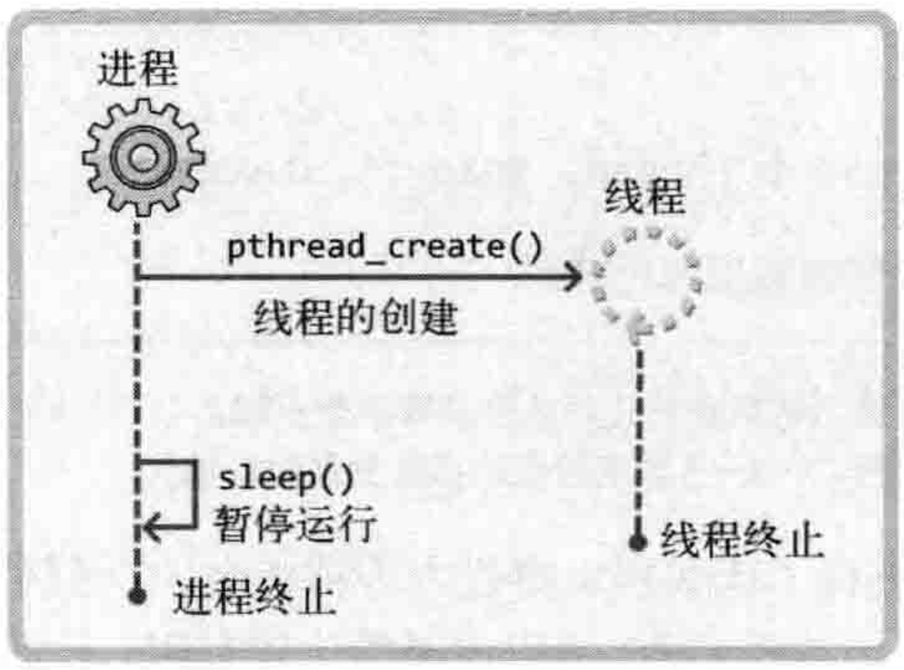

图18-4 示例thread1.c的执行流程

图18-4中的虚线代表执行流称，向下的箭头指的是执行流，横向箭头是函数调用。这些都是简单的符号，可以结合示例理解。接下来将上述示例的第15行sleep函数的调用语句改成如下形式：

```
sleep(2);
```

各位运行后可以看到，此时不会像代码中写的那样输出5次“running thread”字符串。因为main函数返回后整个进程将被销毁，如图18-5所示。

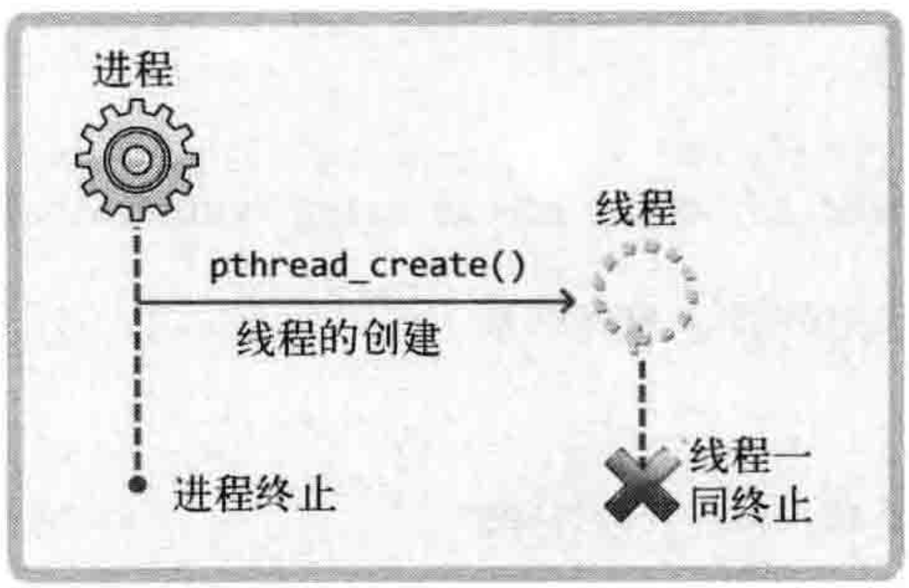

图18-5 终止进程和线程

正因如此，我们在之前的示例中通过调用sleep函数向线程提供了充足的执行时间。

“那线程相关程序中必须适当调用sleep函数！”

并非如此！通过调用sleep函数控制线程的执行相当于预测程序的执行流程，但实际上这是不可能完成的事情。而且稍有不慎，很可能干扰程序的正常执行流。例如，怎么可能在上述示例中准确预测thread_main函数的运行时间，并让main函数恰好等待这么长时间呢？因此，我们不用sleep函数，而是通常利用下面的函数控制线程的执行流。通过下列函数可以更有效地解决现讨论的问题，还可同时了解线程ID的用法。

```
# include <pthread.h>
int pthread_join(pthread_t thread, void ** status);
```

➔ 成功时返回 0，失败时返回其他值。

- thread　该参数值ID的线程终止后才会从该函数返回。
- status　保存线程的main函数返回值的指针变量地址值。

简言之，调用该函数的进程（或线程）将进入等待状态，直到第一个参数为ID的线程终止为止。而且可以得到线程的main函数返回值，所以该函数比较有用。下面通过示例了解该函数的功能。

❖ thread2.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <pthread.h>
5.  void* thread_main(void *arg);
6.
7.  int main(int argc, char *argv[])
8.  {
9.      pthread_t t_id;
10.     int thread_param=5;
11.     void * thr_ret;
12.
13.     if(pthread_create(&t_id, NULL, thread_main, (void*)&thread_param)!=0)
14.     {
15.         puts("pthread_create() error");
16.         return -1;
17.     };
18.
19.     if(pthread_join(t_id, &thr_ret)!=0)
20.     {
21.         puts("pthread_join() error");
22.         return -1;
23.     };
24.
25.     printf("Thread return message: %s \n", (char*)thr_ret);
26.     free(thr_ret);
27.     return 0;
28. }
29.
30. void* thread_main(void *arg)
31. {
32.     int i;
33.     int cnt=*((int*)arg);
34.     char * msg=(char *)malloc(sizeof(char)*50);
35.     strcpy(msg, "Hello, I'am thread~ \n");
36.
37.     for(i=0; i<cnt; i++)
```

```
38.     {
39.         sleep(1); puts("running thread");
40.     }
41.     return (void*)msg;
42. }
```

代码说明

- 第19行：main函数中，针对第13行创建的线程调用pthread_join函数。因此，main函数将等待ID保存在t_id变量中的线程终止。
- 第11、19、41行：希望各位通过这3条语句掌握获取线程的返回值的方法。简言之，第41行返回的值将保存到第19行第二个参数thr_ret。需要注意的是，该返回值是thread_main函数内部动态分配的内存空间地址值。

❖ 运行结果：thread2.c

```
root@my_linux:/tcpip# gcc thread2.c -o tr2 -lpthread
root@my_linux:/tcpip# ./tr2
running thread
running thread
running thread
running thread
running thread
Thread return message: Hello, I'am thread~
```

最后，为了让大家更好地理解该示例，给出其执行流程图，如图18-6所示。请注意观察程序暂停后从线程终止时（线程main函数返回时）重新执行的部分。

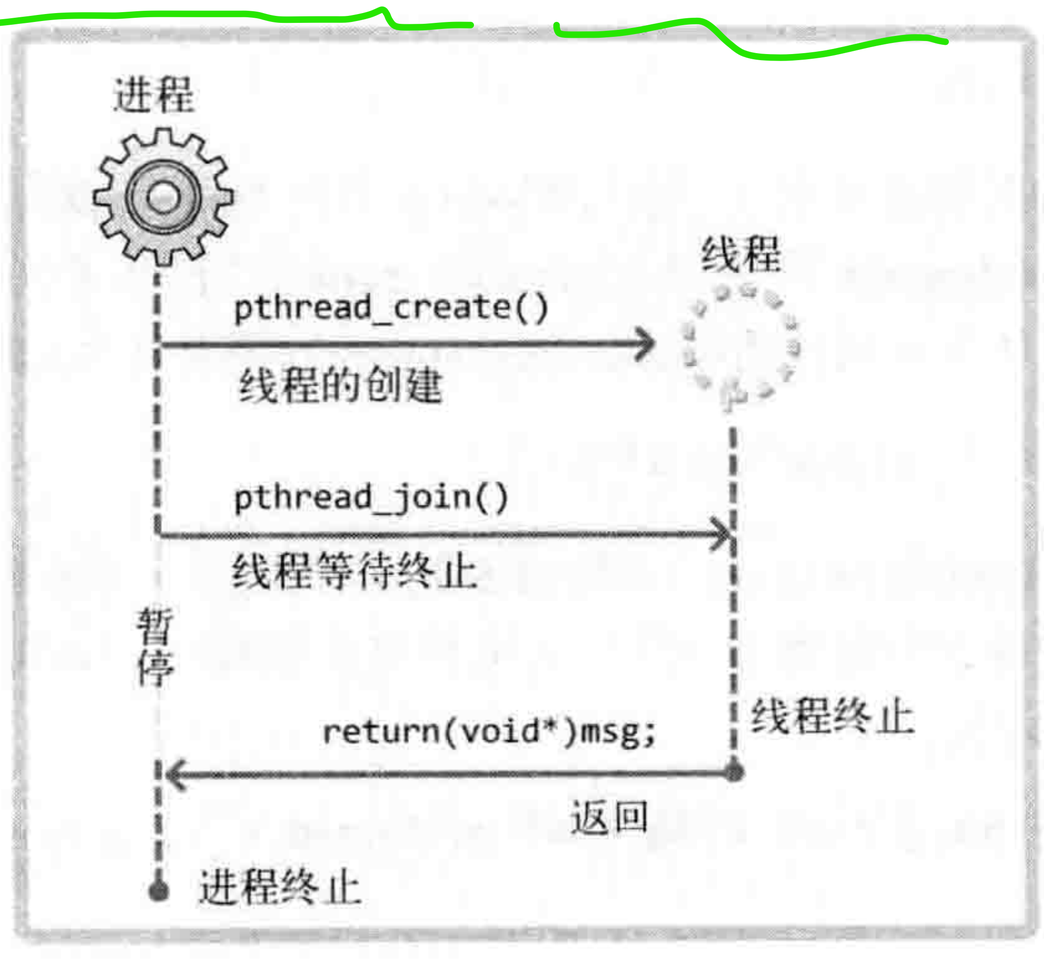

图18-6 调用pthread_join函数

可在临界区内调用的函数

之前的示例中只创建了1个线程，接下来的示例将开始创建多个线程。当然，无论创建多少

线程，其创建方法没有区别。但关于线程的运行需要考虑“多个线程同时调用函数时（执行时）可能产生问题”。这类函数内部存在临界区（Critical Section），也就是说，多个线程同时执行这部分代码时，可能引起问题。临界区中至少存在1条这类代码。

稍后将讨论哪些代码可能成为临界区，多个线程同时执行临界区代码时会产生哪些问题等内容。现阶段只需理解临界区的概念即可。根据临界区是否引起问题，函数可分为以下2类。

- 线程安全函数（Thread-safe function）
- 非线程安全函数（Thread-unsafe function）

线程安全函数被多个线程同时调用时也不会引发问题。反之，非线程安全函数被同时调用时会引发问题。但这并非关于有无临界区的讨论，线程安全的函数中同样可能存在临界区。只是在线程安全函数中，同时被多个线程调用时可通过一些措施避免问题。

幸运的是，大多数标准函数都是线程安全的函数。更幸运的是，我们不用自己区分线程安全的函数和非线程安全的函数（在Windows程序中同样如此）。因为这些平台在定义非线程安全函数的同时，提供了具有相同功能的线程安全的函数。比如，第8章介绍过的如下函数就不是线程安全的函数：

```
struct hostent * gethostbyname(const char * hostname);
```

同时提供线程安全的同一功能的函数。

```
struct hostent * gethostbyname_r(
    const char * name, struct hostent * result, char * buffer, intbuflen,
        int * h_errnop);
```

线程安全函数的名称后缀通常为_r（这与Windows平台不同）。既然如此，多个线程同时访问的代码块中应该调用gethostbyname_r，而不是gethostbyname？当然！但这种方法会给程序员带来沉重的负担。幸好可以通过如下方法自动将gethostbyname函数调用改为gethostbyname_r函数调用！

“声明头文件前定义_REENTRANT宏。”

gethostbyname函数和gethostbyname_r函数的函数名和参数声明都不同，因此，这种宏声明方式拥有巨大的吸引力。另外，无需为了上述宏定义特意添加#define语句，可以在编译时通过添加-D_REENTRANT选项定义宏。

```
root@my_linux:/tcpip# gcc -D_REENTRANT mythread.c -o mthread -lpthread
```

下面编译线程相关代码时均默认添加-D_REENTRANT选项。

工作（Worker）线程模型

之前示例的目的主要是介绍线程概念和创建线程的方法，因此从未涉及1个示例中创建多个

线程的情况。下面给出此类示例。

将要介绍的示例将计算1到10的和，但并不是在main函数中进行累加运算，而是创建2个线程，其中一个线程计算1到5的和，另一个线程计算6到10的和，main函数只负责输出运算结果。这种方式的编程模型称为“工作线程（Worker thread）模型”。计算1到5之和的线程与计算6到10之和的线程将成为main线程管理的工作（Worker）。最后，给出示例代码前先给出程序执行流程图，如图18-7所示。

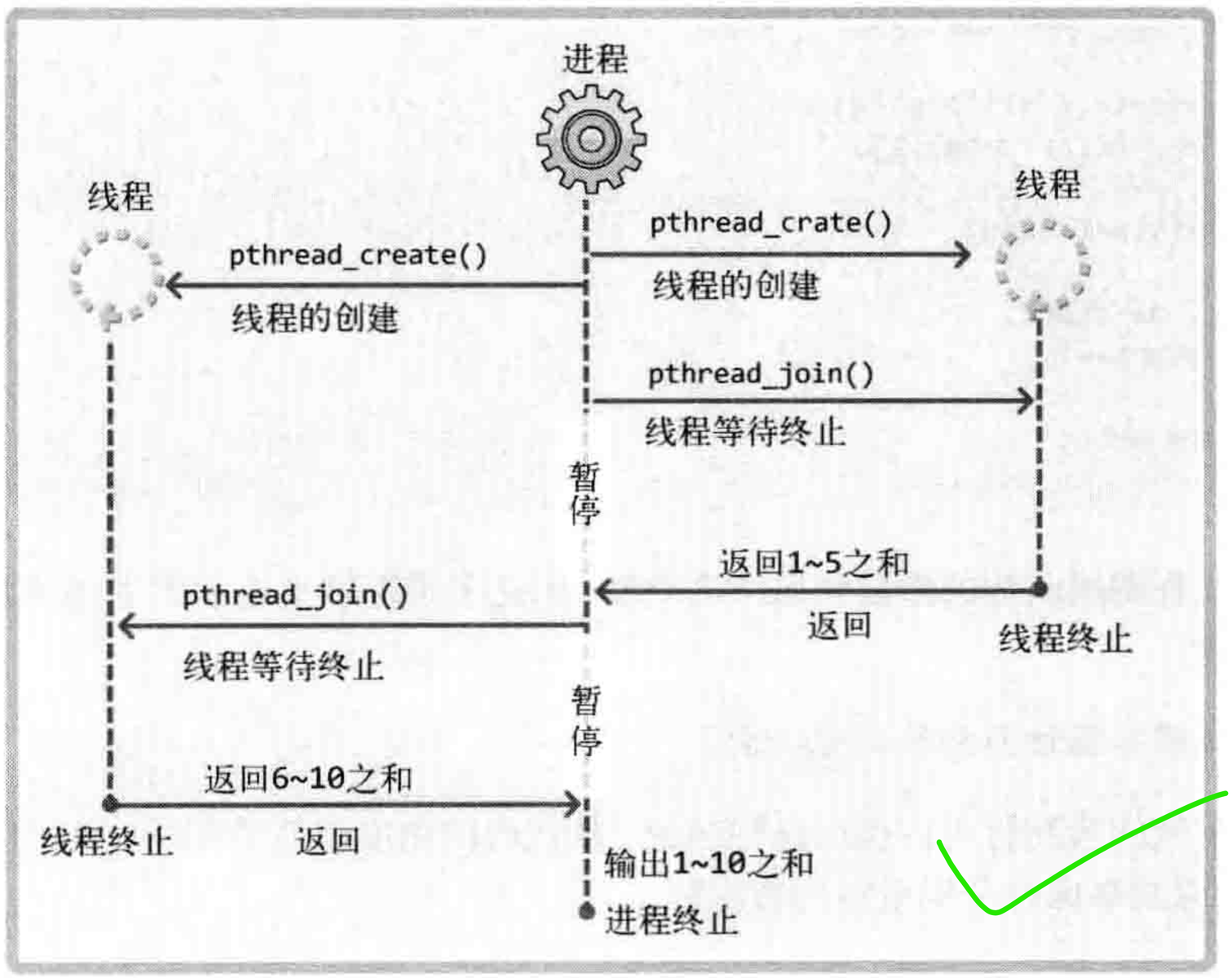

图18-7 示例thread3.c的执行流程

之前也介绍过类似的图，相信各位很容易看懂图18-7描述的内容（只是单纯说明图，并未使用特殊的表示方法）。另外，线程相关代码的执行流程理解起来相对复杂一些，有必要习惯于这类流程图。

❖ thread3.c

```
1.  #include <stdio.h>
2.  #include <pthread.h>
3.  void * thread_summation(void * arg);
4.  int sum=0;
5.
6.  int main(int argc, char *argv[])
7.  {
8.      pthread_t id_t1, id_t2;
9.      int range1[]={1, 5};
10.     int range2[]={6, 10};
```

```
11.
12.     pthread_create(&id_t1, NULL, thread_summation, (void *)range1);
13.     pthread_create(&id_t2, NULL, thread_summation, (void *)range2);
14.
15.     pthread_join(id_t1, NULL);
16.     pthread_join(id_t2, NULL);
17.     printf("result: %d \n", sum);
18.     return 0;
19. }
20.
21. void * thread_summation(void * arg)
22. {
23.     int start=((int*)arg)[0];
24.     int end=((int*)arg)[1];
25.
26.     while(start<=end)
27.     {
28.         sum+=start;
29.         start++;
30.     }
31.     return NULL;
32. }
```

之前讲过线程调用函数的参数和返回值类型，因此不难理解上述示例中创建线程并执行的部分。但需要注意：

“2个线程直接访问全局变量sum!”

通过上述示例的第28行可以得出这种结论。从代码的角度看似乎理所应当，但之所以可行完全是因为2个线程共享保存全局变量的数据区。

❖ 运行结果：thread3.c

```
root@my_linux:/tcpip# gcc thread3.c -D_REENTRANT -o tr3 -lpthread
root@my_linux:/tcpip# ./tr3
result: 55
```

运行结果是55，虽然正确，但示例本身存在问题。此处存在临界区相关问题，因此再介绍另一示例。该示例与上述示例相似，只是增加了发生临界区相关错误的可能性，即使在高配置系统环境下也容易验证产生的错误。

❖ thread4.c

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <stdlib.h>
4.  #include <pthread.h>
5.  #define NUM_THREAD 100
```

```
6.
7.  void * thread_inc(void * arg);
8.  void * thread_des(void * arg);
9.  long long num=0;    // long long类型是64位整数型
10.
11. int main(int argc, char *argv[])
12. {
13.     pthread_t thread_id[NUM_THREAD];
14.     int i;
15.
16.     printf("sizeof long long: %d \n", sizeof(long long));    // 查看long long的大小
17.     for(i=0; i<NUM_THREAD; i++)
18.     {
19.         if(i%2)
20.             pthread_create(&(thread_id[i]), NULL, thread_inc, NULL);
21.         else
22.             pthread_create(&(thread_id[i]), NULL, thread_des, NULL);
23.     }
24.
25.     for(i=0; i<NUM_THREAD; i++)
26.         pthread_join(thread_id[i], NULL);
27.
28.     printf("result: %lld \n", num);
29.     return 0;
30. }
31.
32. void * thread_inc(void * arg)
33. {
34.     int i;
35.     for(i=0; i<50000000; i++)
36.         num+=1;
37.     return NULL;
38. }
39. void * thread_des(void * arg)
40. {
41.     int i;
42.     for(i=0; i<50000000; i++)
43.         num-=1;
44.     return NULL;
45p.}
```

上述示例中共创建了100个线程，其中一半执行thread_inc函数中的代码，另一半则执行thread_des函数中的代码。全局变量num经过增减过程后应存有0，通过运行结果观察是否真能得到。

❖ 运行结果：thread4.c

```
root@my_linux:/tcpip# gcc thread4.c -D_REENTRANT -o th4 -lpthread
root@my_linux:/tcpip# ./th4
sizeof long long: 8
result: 4284144869
```

```
root@my_linux:/tcpip# ./th4
sizeof long long: 8
result: 8577052432
root@my_linux:/tcpip# ./th4
sizeof long long: 8
result: 21446758095
```

运行结果并不是0！而且每次运行的结果均不同。虽然其原因尚不得而知，但可以肯定的是，这对于线程的应用是个大问题。

18.3　线程存在的问题和临界区

我们还不知道示例thread4.c中产生问题的原因，下面分析该问题并给出解决方案。

多个线程访问同一变量是问题

示例thread4.c的问题如下：

“2个线程正在同时访问全局变量num。”

此处的“访问”是指值的更改。产生问题的原因可能还有很多，因此需要准确理解。虽然示例中访问的对象是全局变量，但这并非全局变量引发的问题。任何内存空间——只要被同时访问——都可能发生问题。

“不是说线程会分时使用CPU吗？那应该不会出现同时访问变量的情况啊。”

当然，此处的“同时访问”与各位所想的有一定区别。下面通过示例解释“同时访问”的含义，并说明为何会引起问题。假设2个线程要执行将变量值逐次加1的工作，如图18-8所示。

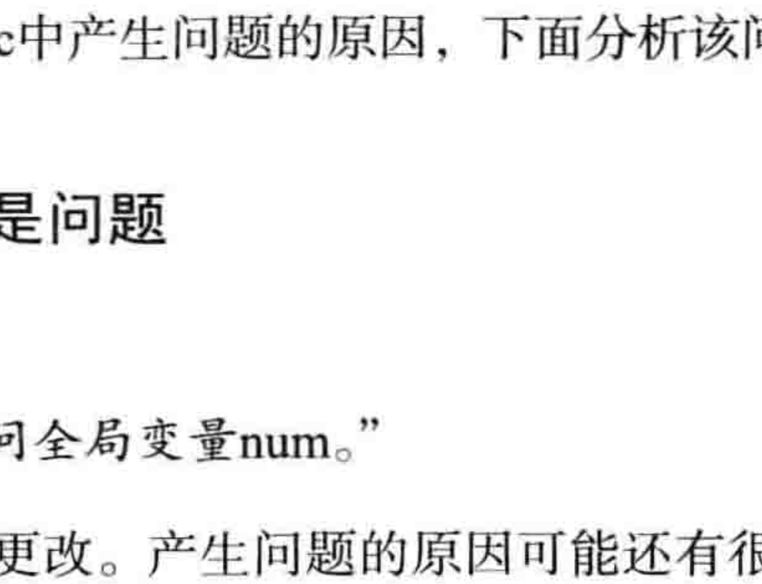

图18-8　等待中的2个线程

图18-8中描述的是2个线程准备将变量num的值加1的情况。在此状态下，线程1将变量num的

值增加到100后，线程2再访问num时，变量num中将按照我们的预想保存101。图18-9是线程1将变量num完全增加后的情形。

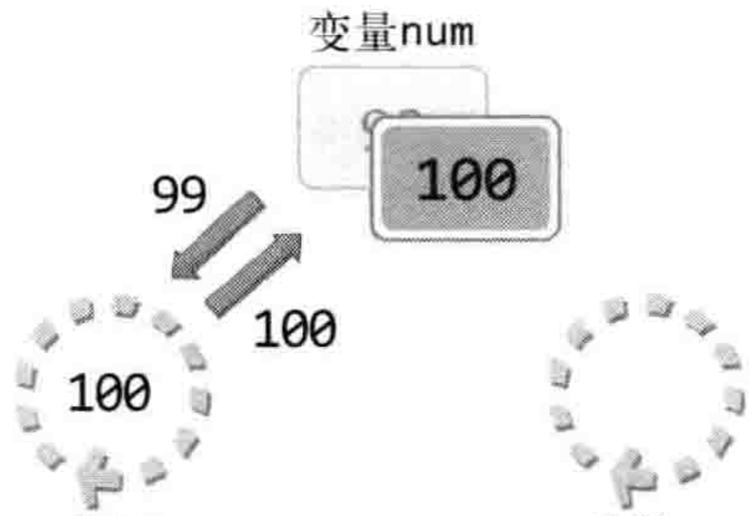

图18-9 线程的加法运算1-1

图18-9中需要注意值的增加方式，值的增加需要CPU运算完成，变量num中的值不会自动增加。线程1首先读该变量的值并将其传递到CPU，获得加1之后的结果100，最后再把结构写回变量num，这样num中就保存100。接下来给出线程2的执行过程，如图18-10所示。

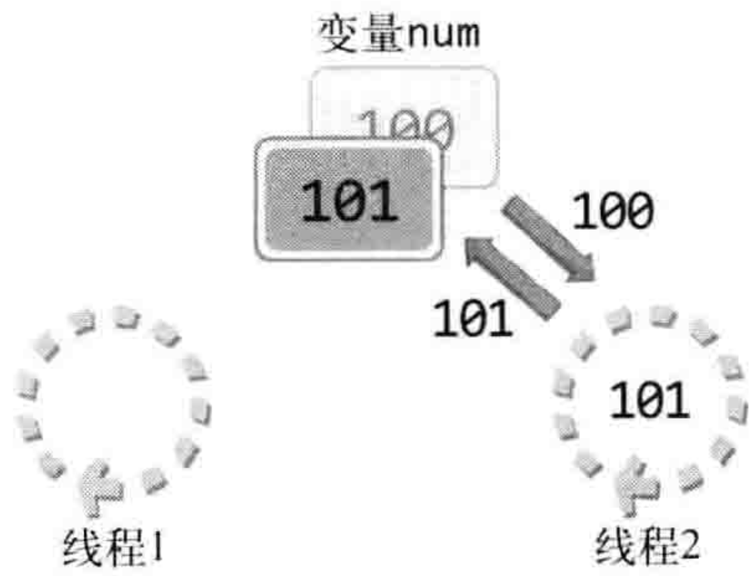

图18-10 线程的加法运算 1-2

变量num中将保存101，但这是最理想的情况。线程1完全增加num值之前，线程2完全有可能通过切换得到CPU资源。下面从头再来。图18-11描绘的是线程1读取变量num的值并完成加1运算时的情况，只是加1后的结果尚未写入变量num。

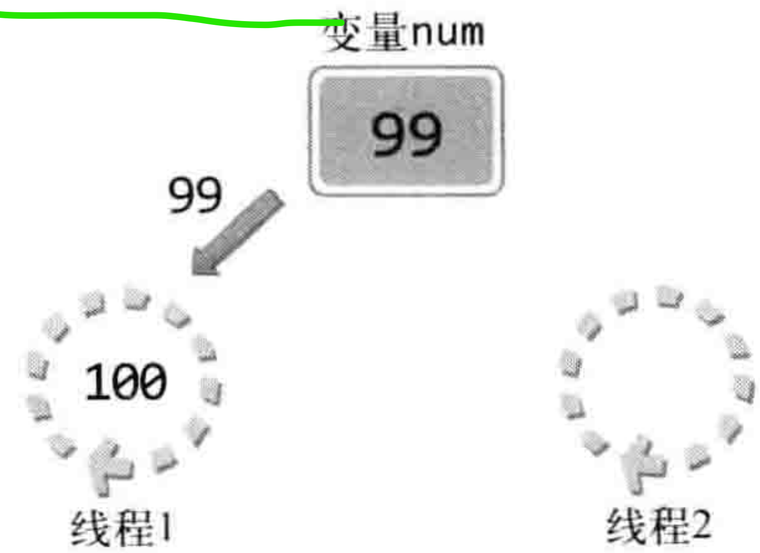

图18-11 线程的加法运算2-1

接下来就要将100保存到变量num中，但执行该操作前，执行流程跳转到了线程2。幸运的是（是否真正幸运稍后再论），线程2完成了加1运算，并将加1之后的结果写入变量num，如图18-12所示。

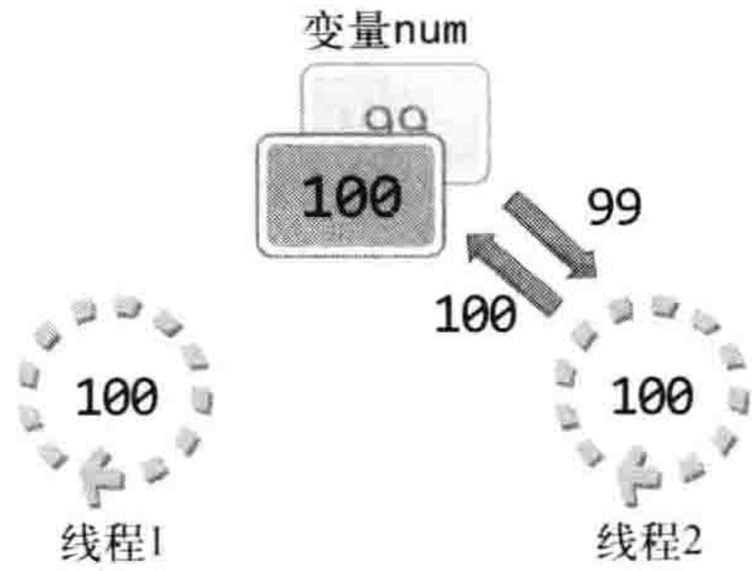

图18-12　线程的加法运算 2-2

从图18-12中可以看到，变量num的值尚未被线程1加到100，因此线程2读到的变量num的值为99，结果是线程2将num值改成100。还剩下线程1将运算后的值写入变量num的操作。接下来给出该过程，如图18-13所示。

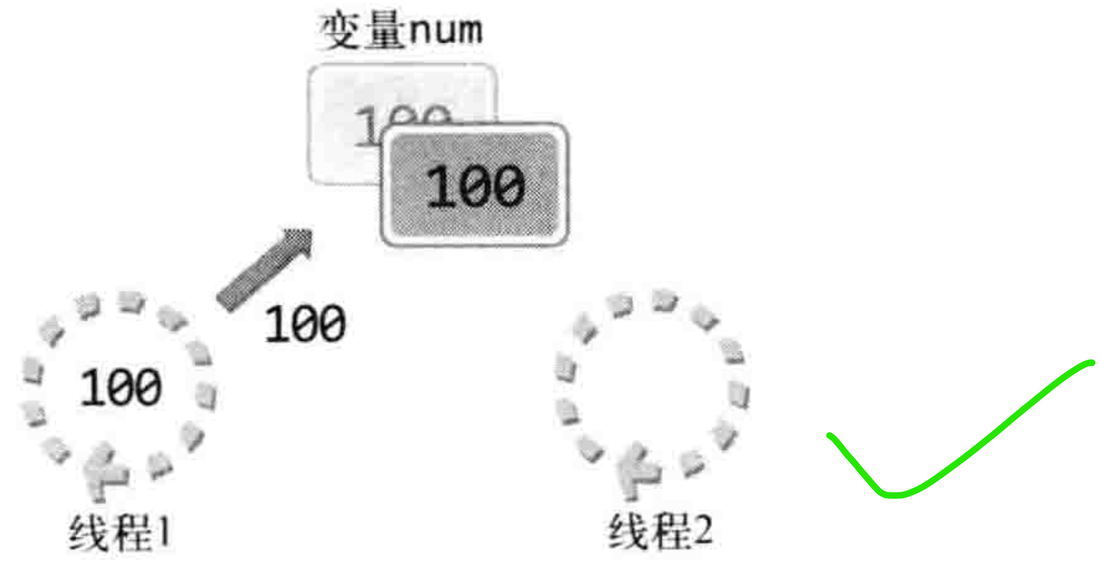

图18-13　线程的加法运算 2-3

很可惜，此时线程1将自己的运算结果100再次写入变量num，结果变量num变成100。虽然线程1和线程2各做了1次加1运算，却得到了意想不到的结果。因此，线程访问变量num时应该阻止其他线程访问，直到线程1完成运算。这就是同步（Synchronization）。相信各位也意识到了多线程编程中“同步”的必要性，且能够理解thread4.c的运行结果。

临界区位置

划分临界区并不难。既然临界区定义为如下这种形式，那就在示例thread4.c中寻找。

“函数内同时运行多个线程时引起问题的多条语句构成的代码块。”

全局变量num是否应该视为临界区？不是！因为它不是引起问题的语句。该变量并非同时运

行的语句，只是代表内存区域的声明而已。临界区通常位于由线程运行的函数内部。下面观察示例thread4.c中的2个main函数。

```
void * thread_inc(void * arg)
{
    int i;
    for(i = 0; i < 50000000; i++)
        num += 1;  // 临界区
    return NULL;
}
```

```
void * thread_des(void * arg)
{
    int i;
    for(i = 0; i < 50000000; i++)
        num -= 1;   // 临界区
    return NULL;
}
```

由代码注释可知，临界区并非num本身，而是访问num的2条语句。这2条语句可能由多个线程同时运行，也是引起问题的直接原因。产生的问题可以整理为如下3种情况。

- 2个线程同时执行thread_inc函数。
- 2个线程同时执行thread_des函数。
- 2个线程分别执行thread_inc函数和thread_des函数。

需要关注最后一点，它意味着如下情况下也会引发问题：

> “线程1执行thread_inc函数的num+=1语句的同时，线程2执行thread_des函数的num-=1语句。”

也就是说，2条不同语句由不同线程同时执行时，也有可能构成临界区。前提是这2条语句访问同一内存空间。

18.4 线程同步

前面探讨了线程中存在的问题，接下来就要讨论解决方法——线程同步。

同步的两面性

线程同步用于解决线程访问顺序引发的问题。需要同步的情况可以从如下两方面考虑。

- 同时访问同一内存空间时发生的情况。
- 需要指定访问同一内存空间的线程执行顺序的情况。

之前已解释过前一种情况，因此重点讨论第二种情况。这是“控制（Control）线程执行顺序”的相关内容。假设有A、B两个线程，线程A负责向指定内存空间写入（保存）数据，线程B负责取走该数据。这种情况下，线程A首先应该访问约定的内存空间并保存数据。万一线程B先访问并取走数据，将导致错误结果。像这种需要控制执行顺序的情况也需要使用同步技术。

稍后将介绍“互斥量”（Mutex）和“信号量”（Semaphore）这2种同步技术。二者概念上十分接近，只要理解了互斥量就很容易掌握信号量。而且大部分同步技术的原理都大同小异，因此，只要掌握了本章介绍的同步技术，就很容易掌握并运用Windows平台下的同步技术。

互斥量

互斥量是“Mutual Exclusion”的简写，表示不允许多个线程同时访问。互斥量主要用于解决线程同步访问的问题。为了理解好互斥量，请观察如下对话过程。

- 东秀：“请问里面有人吗？”
- 英秀：“是的，有人。”

- 东秀：“您好！”
- 英秀：“请稍等！”

相信各位也猜到了上述对话发生的场景。现实世界中的临界区就是洗手间。洗手间无法同时容纳多人（比作线程），因此可以将临界区比喻为洗手间。而且这里发生的所有事情几乎可以全部套用到临界区同步过程。洗手间使用规则如下。

- 为了保护个人隐私，进洗手间时锁上门，出来时再打开。
- 如果有人使用洗手间，其他人需要在外面等待。
- 等待的人数可能很多，这些人需排队进入洗手间。

这就是洗手间的使用规则。同样，线程中为了保护临界区也需要套用上述规则。洗手间中存在，但之前的线程示例中缺少的是什么呢？就是锁机制。线程同步中同样需要锁，就像洗手间示例中使用的那样。互斥量就是一把优秀的锁，接下来介绍互斥量的创建及销毁函数。

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_ mutexattr_t * attr);
int pthread_mutex_destroy(pthread_mutex_t * mutex);
```

➔ 成功时返回 0，失败时返回其他值。

mutex	创建互斥量时传递保存互斥量的变量地址值，销毁时传递需要销毁的互斥量地址值。
attr	传递即将创建的互斥量属性，没有特别需要指定的属性时传递NULL。

从上述函数声明中也可看出，为了创建相当于锁系统的互斥量，需要声明如下pthread_mutex_t型变量：

```
pthread_mutex_t mutex;
```

该变量的地址将传递给pthread_mutex_init函数，用来保存操作系统创建的互斥量（锁系统）。调用pthread_mutex_destroy函数时同样需要该信息。如果不需要配置特殊的互斥量属性，则向第二个参数传递NULL时，可以利用PTHREAD_MUTEX_INITIALIZER宏进行如下声明：

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

但推荐各位尽可能使用pthread_mutex_init函数进行初始化，因为通过宏进行初始化时很难发现发生的错误。接下来介绍利用互斥量锁住或释放临界区时使用的函数。

```
#include <pthread.h>

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

➔ 成功时返回0，失败时返回其他值。

函数名本身含有lock、unlock等词汇，很容易理解其含义。进入临界区前调用的函数就是pthread_mutex_lock。调用该函数时，发现有其他线程已进入临界区，则pthread_mutex_lock函数不会返回，直到里面的线程调用pthread_mutex_unlock函数退出临界区为止。也就是说，其他线程让出临界区之前，当前线程将一直处于阻塞状态。接下来整理一下保护临界区的代码块编写方法。创建好互斥量的前提下，可以通过如下结构保护临界区。

```
pthread_mutex_lock(&mutex);
// 临界区的开始
// . . . . .
// 临界区的结束
pthread_mutex_unlock(&mutex);
```

简言之，就是利用lock和unlock函数围住临界区的两端。此时互斥量相当于一把锁，阻止多个线程同时访问。还有一点需要注意，线程退出临界区时，如果忘了调用pthread_mutex_unlock函数，那么其他为了进入临界区而调用pthread_mutex_lock函数的线程就无法摆脱阻塞状态。这

种情况称为“死锁”（Dead-lock），需要格外注意。接下来利用互斥量解决示例thread4.c中遇到的问题。

❖ mutex.c

```
#include <"与示例thread4.c的头声明一致。">
#define NUM_THREAD  100
void * thread_inc(void * arg);
void * thread_des(void * arg);

long long num=0;
pthread_mutex_t mutex;

int main(int argc, char *argv[])
{
    pthread_t thread_id[NUM_THREAD];
    int i;

    pthread_mutex_init(&mutex, NULL);

    for(i=0; i<NUM_THREAD; i++)
    {
        if(i%2)
            pthread_create(&(thread_id[i]), NULL, thread_inc, NULL);
        else
            pthread_create(&(thread_id[i]), NULL, thread_des, NULL);
    }

    for(i=0; i<NUM_THREAD; i++)
        pthread_join(thread_id[i], NULL);

    printf("result: %lld \n", num);
    pthread_mutex_destroy(&mutex);
    return 0;
}

void * thread_inc(void * arg)
{
    int i;
    pthread_mutex_lock(&mutex);
    for(i=0; i<50000000; i++)
        num+=1;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
void * thread_des(void * arg)
{
    int i;
    for(i=0; i<50000000; i++)
    {
        pthread_mutex_lock(&mutex);
        num-=1;
```

```
48.         pthread_mutex_unlock(&mutex);
49.     }
50.     return NULL;
51. }
```

代码说明

- 第7行：声明了保存互斥量读取值的变量。之所以声明全局变量是因为，thread_inc函数和thread_des函数都需要访问互斥量。
- 第28行：销毁互斥量。不需要互斥量时应该销毁。
- 第35、38行：实际临界区只是第37行。但此处连同第36行的循环语句一起用作临界区，调用了lock、unlock函数。关于这一点稍后再讨论。
- 第46、48行：通过lock、unlock函数围住对应于临界区的第47行语句。

❖ 运行结果：mutex.c

```
root@my_linux:/tcpip# gcc mutex.c -D_REENTRANT -o mutex -lpthread
root@my_linux:/tcpip# ./mutex
result: 0
```


从运行结果可以看出，已解决了示例thread4.c中的问题。但确认运行结果需要等待较长时间。因为互斥量lock、unlock函数的调用过程要比想象中花费更长时间。首先分析一下thread_inc函数的同步过程。

```
void * thread_inc(void * arg)
{
    int i;
    pthread_mutex_lock(&mutex);
    for(i = 0; i < 50000000; i++)
        num += 1;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

以上临界区划分范围较大，但这是考虑到如下优点所做的决定：

“最大限度减少互斥量lock、unlock函数的调用次数。”

上述示例中，thread_des函数比thread_inc函数多调用49,999,999次互斥量lock、unlock函数，表现出人可以感知的速度差异。如果不太关注线程的等待时间，可以适当扩展临界区。但变量num的值增加到50,000,000前不允许其他线程访问，这反而成了缺点。其实这里没有正确答案，需要根据不同程序酌情考虑究竟扩大还是缩小临界区。此处没有公式可言，各位需要培养自己的判断能力。

信号量

下面介绍信号量。信号量与互斥量极为相似，在互斥量的基础上很容易理解信号量。此处只涉及利用“二进制信号量”（只用0和1）完成“控制线程顺序”为中心的同步方法。下面给出信号量创建及销毁方法。

```
#include <semaphore.h>

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

➔ 成功时返回 0，失败时返回其他值。

- sem　创建信号量时传递保存信号量的变量地址值，销毁时传递需要销毁的信号量变量地址值。
- pshared　传递其他值时，创建可由多个进程共享的信号量；传递0时，创建只允许1个进程内部使用的信号量。我们需要完成同一进程内的线程同步，故传递0。
- value　指定新创建的信号量初始值。

上述函数的pshared参数超出了我们关注的范围，故默认向其传递0。稍后讲解通过value参数初始化的信号量值究竟是多少。接下来介绍信号量中相当于互斥量lock、unlock的函数。

```
#include <semaphore.h>

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

➔ 成功时返回 0，失败时返回其他值。

- sem　传递保存信号量读取值的变量地址值，传递给sem_post时信号量增1，传递给sem_wait时信号量减1。

调用sem_init函数时，操作系统将创建信号量对象，此对象中记录着“信号量值”（Semaphore Value）整数。该值在调用sem_post函数时增1，调用sem_wait函数时减1。但信号量的值不能小于0，因此，在信号量为0的情况下调用sem_wait函数时，调用函数的线程将进入阻塞状态（因为函数未返回）。当然，此时如果有其他线程调用sem_post函数，信号量的值将变为1，而原本阻塞的线程可以将该信号量重新减为0并跳出阻塞状态。实际上就是通过这种特性完成临界区的同步操作，可以通过如下形式同步临界区（假设信号量的初始值为1）。

```
sem_wait(&sem);  // 信号量变为 0. . .
// 临界区的开始
// . . . . . .
// 临界区的结束
```

```
sem_post(&sem);  // 信号量变为1. . .
```

上述代码结构中，调用sem_wait函数进入临界区的线程在调用sem_post函数前不允许其他线程进入临界区。信号量的值在0和1之间跳转，因此，具有这种特性的机制称为“二进制信号量”。接下来给出信号量相关示例。即将介绍的示例并非关于同时访问的同步，而是关于控制访问顺序的同步。该示例的场景如下：

> “线程A从用户输入得到值后存入全局变量num，此时线程B将取走该值并累加。该过程共进行5次，完成后输出总和并退出程序。”

为了按照上述要求构建程序，应按照线程A、线程B的顺序访问变量num，且需要线程同步。接下来给出示例，分析该示例可能需要花费一定时间。

❖ semaphore.c

```
1.  #include <stdio.h>
2.  #include <pthread.h>
3.  #include <semaphore.h>
4.
5.  void * read(void * arg);
6.  void * accu(void * arg);
7.  static sem_t sem_one;
8.  static sem_t sem_two;
9.  static int num;
10.
11. int main(int argc, char *argv[])
12. {
13.     pthread_t id_t1, id_t2;
14.     sem_init(&sem_one, 0, 0);
15.     sem_init(&sem_two, 0, 1);
16.
17.     pthread_create(&id_t1, NULL, read, NULL);
18.     pthread_create(&id_t2, NULL, accu, NULL);
19.
20.     pthread_join(id_t1, NULL);
21.     pthread_join(id_t2, NULL);
22.
23.     sem_destroy(&sem_one);
24.     sem_destroy(&sem_two);
25.     return 0;
26. }
27.
28. void * read(void * arg)
29. {
30.     int i;
31.     for(i=0; i<5; i++)
32.     {
33.         fputs("Input num: ", stdout);
34.
35.         sem_wait(&sem_two);
```

```
36.          scanf("%d", &num);
37.          sem_post(&sem_one);
38.      }
39.      return NULL;
40. }
41. void * accu(void * arg)
42. {
43.      int sum=0, i;
44.      for(i=0; i<5; i++)
45.      {
46.          sem_wait(&sem_one);
47.          sum+=num;
48.          sem_post(&sem_two);
49.      }
50.      printf("Result: %d \n", sum);
51.      return NULL;
52. }
```

代码说明

- 第14、15行：生成2个信号量，一个信号量的值为0，另一个为1。一定要掌握需要2个信号量的原因。
- 第35、48行：利用信号量变量sem_two调用wait函数和post函数。这是为了防止在调用accu函数的线程还未取走数据的情况下，调用read函数的线程覆盖原值。
- 第37、46行：利用信号量变量sem_one调用wait和post函数。这是为了防止调用read函数的线程写入新值前，accu函数取走（再取走旧值）数据。

❖ 运行结果：semaphore.c

```
root@my_linux:/tcpip# gcc semaphore.c -D_REENTRANT -o sema -lpthread
root@my_linux:/tcpip# ./sema
Input num: 1
Input num: 2
Input num: 3
Input num: 4
Input num: 5
Result: 15
```

如果各位还不太理解为何需要2个信号量，可将代码中的注释部分去掉，再运行程序并观察运行结果。以上就是线程相关的全部理论知识，下面在此基础上编写服务器端。

18.5　线程的销毁和多线程并发服务器端的实现

我们之前只讨论了线程的创建和控制，而线程的销毁同样重要。下面先介绍线程的销毁，再实现多线程服务器端。

销毁线程的3种方法

Linux线程并不是在首次调用的线程main函数返回时自动销毁，所以用如下2种方法之一加以明确。否则由线程创建的内存空间将一直存在。

- 调用pthread_join函数。
- 调用pthread_detach函数。

之前调用过pthread_join函数。调用该函数时，不仅会等待线程终止，还会引导线程销毁。但该函数的问题是，线程终止前，调用该函数的线程将进入阻塞状态。因此，通常通过如下函数调用引导线程销毁。

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

➔ 成功时返回0，失败时返回其他值。

- thread　终止的同时需要销毁的线程ID。

调用上述函数不会引起线程终止或进入阻塞状态，可以通过该函数引导销毁线程创建的内存空间。调用该函数后不能再针对相应线程调用pthread_join函数，这需要格外注意。虽然还有方法在创建线程时可以指定销毁时机，但与pthread_detach方式相比，结果上没有太大差异，故省略其说明。在下面的多线程并发服务器端的实现过程中，希望各位同样关注线程销毁的部分。

多线程并发服务器端的实现

本节并不打算介绍回声服务器端，而是介绍多个客户端之间可以交换信息的简单的聊天程序。希望各位通过本示例复习线程的使用方法及同步的处理方法，还可以再次思考临界区的处理方式。

无论服务器端还是客户端，代码量都不少，故省略可以从其他示例中得到或从源代码中下载的头文件声明。同时最大程度地减少异常处理的代码。

❖ chat_server.c

```
1.  #include <"头文件声明请参考源文件。">
2.  #define BUF_SIZE 100
3.  #define MAX_CLNT 256
4.
5.  void * handle_clnt(void * arg);
6.  void send_msg(char * msg, int len);
7.  void error_handling(char * msg);
8.
```

```
int clnt_cnt=0;
int clnt_socks[MAX_CLNT];
pthread_mutex_t mutx;

int main(int argc, char *argv[])
{
    int serv_sock, clnt_sock;
    struct sockaddr_in serv_adr, clnt_adr;
    int clnt_adr_sz;
    pthread_t t_id;
    if(argc!=2) {
        printf("Usage : %s <port>\n", argv[0]);
        exit(1);
    }

    pthread_mutex_init(&mutx, NULL);
    serv_sock=socket(PF_INET, SOCK_STREAM, 0);

    memset(&serv_adr, 0, sizeof(serv_adr));
    serv_adr.sin_family=AF_INET;
    serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
    serv_adr.sin_port=htons(atoi(argv[1]));

    if(bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr))==-1)
        error_handling("bind() error");
    if(listen(serv_sock, 5)==-1)
        error_handling("listen() error");

    while(1)
    {
        clnt_adr_sz=sizeof(clnt_adr);
        clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_adr,&clnt_adr_sz);

        pthread_mutex_lock(&mutx);
        clnt_socks[clnt_cnt++]=clnt_sock;
        pthread_mutex_unlock(&mutx);

        pthread_create(&t_id, NULL, handle_clnt, (void*)&clnt_sock);
        pthread_detach(t_id);
        printf("Connected client IP: %s \n", inet_ntoa(clnt_adr.sin_addr));
    }
    close(serv_sock);
    return 0;
}

void * handle_clnt(void * arg)
{
    int clnt_sock=*((int*)arg);
    int str_len=0, i;
    char msg[BUF_SIZE];

    while((str_len=read(clnt_sock, msg, sizeof(msg)))!=0)
        send_msg(msg, str_len);

```

```
63.     pthread_mutex_lock(&mutx);
64.     for(i=0; i<clnt_cnt; i++) // remove disconnected client
65.     {
66.         if(clnt_sock==clnt_socks[i])
67.         {
68.             while(i++<clnt_cnt-1)
69.                 clnt_socks[i]=clnt_socks[i+1];
70.             break;
71.         }
72.     }
73.     clnt_cnt--;
74.     pthread_mutex_unlock(&mutx);
75.     close(clnt_sock);
76.     return NULL;
77. }
78. void send_msg(char * msg, int len) // send to all
79. {
80.     int i;
81.     pthread_mutex_lock(&mutx);
82.     for(i=0; i<clnt_cnt; i++)
83.         write(clnt_socks[i], msg, len);
84.     pthread_mutex_unlock(&mutx);
85. }
86. void error_handling(char * msg)
87. {
88.     //与之前示例的error_handling函数一致。
89. }
```


代码说明

- 第9、10行：用于管理接入的客户端套接字的变量和数组。访问这2个变量的代码将构成临界区。
- 第43行：每当有新连接时，将相关信息写入变量clnt_cnt和clnt_socks。
- 第46行：创建线程向新接入的客户端提供服务。由该线程执行第53行定义的函数。
- 第47行：调用pthread_detach函数从内存中完全销毁已终止的线程。
- 第78行：该函数负责向所有连接的客户端发送消息。

上述示例中，各位必须掌握的并不是聊天服务器端的实现方式，而是临界区的构成形式。上述示例中的临界区具有如下特点：

“访问全局变量clnt_cnt和数组clnt_socks的代码将构成临界区！”

添加或删除客户端时，变量clnt_cnt和数组clnt_socks同时发生变化。因此，在如下情形中均会导致数据不一致，从而引发严重错误。

- ❑ 线程A从数组clnt_socks中删除套接字信息，同时线程B读取clnt_cnt变量。
- ❑ 线程A读取变量clnt_cnt，同时线程B将套接字信息添加到clnt_socks数组。

因此，如上述示例所示，访问变量clnt_cnt和数组clnt_socks的代码应组织在一起并构成临界区。大家现在应该对我之前说过的这句话有同感了吧：

“此处的‘访问’是指值的更改。产生问题的原因可能还有很多，因此需要准确理解。”

接下来介绍聊天客户端，客户端示例为了分离输入和输出过程而创建了线程。代码分析并不难，故省略源代码相关说明。

❖ chat_clnt.c

```
1.  #include <"头文件声明请参考源文件。">
2.  #define BUF_SIZE 100
3.  #define NAME_SIZE 20
4.
5.  void * send_msg(void * arg);
6.  void * recv_msg(void * arg);
7.  void error_handling(char * msg);
8.
9.  char name[NAME_SIZE]="[DEFAULT]";
10. char msg[BUF_SIZE];
11.
12. int main(int argc, char *argv[])
13. {
14.     int sock;
15.     struct sockaddr_in serv_addr;
16.     pthread_t snd_thread, rcv_thread;
17.     void * thread_return;
18.     if(argc!=4) {
19.         printf("Usage : %s <IP> <port> <name>\n", argv[0]);
20.         exit(1);
21.     }
22.
23.     sprintf(name, "[%s]", argv[3]);
24.     sock=socket(PF_INET, SOCK_STREAM, 0);
25.
26.     memset(&serv_addr, 0, sizeof(serv_addr));
27.     serv_addr.sin_family=AF_INET;
28.     serv_addr.sin_addr.s_addr=inet_addr(argv[1]);
29.     serv_addr.sin_port=htons(atoi(argv[2]));
30.
31.     if(connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))==-1)
32.         error_handling("connect() error");
33.
34.     pthread_create(&snd_thread, NULL, send_msg, (void*)&sock);
35.     pthread_create(&rcv_thread, NULL, recv_msg, (void*)&sock);
36.     pthread_join(snd_thread, &thread_return);
37.     pthread_join(rcv_thread, &thread_return);
38.     close(sock);
39.     return 0;
40. }
41.
42. void * send_msg(void * arg)    // send thread main
43. {
44.     int sock=*((int*)arg);
45.     char name_msg[NAME_SIZE+BUF_SIZE];
```

```
46.     while(1)
47.     {
48.         fgets(msg, BUF_SIZE, stdin);
49.         if(!strcmp(msg,"q\n")||!strcmp(msg,"Q\n"))
50.         {
51.             close(sock);
52.             exit(0);
53.         }
54.         sprintf(name_msg,"%s %s", name, msg);
55.         write(sock, name_msg, strlen(name_msg));
56.     }
57.     return NULL;
58. }
59. void * recv_msg(void * arg)    // read thread main
60. {
61.     int sock=*((int*)arg);
62.     char name_msg[NAME_SIZE+BUF_SIZE];
63.     int str_len;
64.     while(1)
65.     {
66.         str_len=read(sock, name_msg, NAME_SIZE+BUF_SIZE-1);
67.         if(str_len==-1)
68.             return (void*)-1;
69.         name_msg[str_len]=0;
70.         fputs(name_msg, stdout);
71.     }
72.     return NULL;
73. }
74.
75. void error_handling(char *msg)
76. {
77.     //与之前示例的error_handling函数一致。
78. }
```

下面给出运行结果。接入服务器端的客户端IP均为127.0.0.1，因为服务器端和客户端均在同一台计算机中运行。

❖ 运行结果：chat_server.c

```
root@my_linux:/tcpip# gcc chat_serv.c -D_REENTRANT -o cserv -lpthread
root@my_linux:/tcpip# ./cserv 9190
Connected client IP: 127.0.0.1
Connected client IP: 127.0.0.1
Connected client IP: 127.0.0.1
```

❖ 运行结果：chat_clnt.c One [From Yoon]

```
root@my_linux:/tcpip# gcc chat_clnt.c -D_REENTRANT -o cclnt -lpthread
root@my_linux:/tcpip# ./cclnt 127.0.0.1 9190 Yoon
Hi everyone~
```

```
[Yoon] Hi everyone~
[Choi] Hi Yoon
[Hong] Hi~ friends
```

❖ 运行结果：chat_clnt.c Two [From Choi]

```
root@my_linux:/tcpip# ./cclnt 127.0.0.1 9190 Choi
[Yoon] Hi everyone~
Hi Yoon
[Choi] Hi Yoon
[Hong] Hi~ friends
```

❖ 运行结果：chat_clnt.c Three [From Hong]

```
root@my_linux:/tcpip# ./cclnt 127.0.0.1 9190 Hong
[Yoon] Hi everyone~
[Choi] Hi Yoon
Hi~ friends
[Hong] Hi~ friends
```

18.6　习题

(1) 单CPU系统中如何同时执行多个进程？请解释该过程中发生的上下文切换。

(2) 为何线程的上下文切换速度相对更？线程间数据交换为何不需要类似IPC的特别技术？

(3) 请从执行流角度说明进程和线程的区别。

(4) 下列关于临界区的说法错误的是？

a. 临界区是多个线程同时访问时发生问题的区域。
b. 线程安全的函数中不存在临界区，即便多个线程同时调用也不会发生问题。
c. 1个临界区只能由1个代码块，而非多个代码块构成。换言之，线程A执行的代码块A和线程B执行的代码块B之间绝对不会构成临界区。
d. 临界区由访问全局变量的代码构成。其他变量中不会发生问题。

(5) 下列关于线程同步的描述错误的是？

a. 线程同步就是限制访问临界区。
b. 线程同步也具有控制线程执行顺序的含义。
c. 互斥量和信号量是典型的同步技术。
d. 线程同步是代替进程IPC的技术。

(6) 请说明完全销毁Linux线程的2种方法。

(7) 请利用多线程技术实现回声服务器端，但要让所有线程共享保存客户端消息的内存空间（char数组）。这么做只是为了应用本章的同步技术，其实不符合常理。

(8) 上一题要求所有线程共享保存回声消息的内存空间，如果采用这种方式，无论是否同步都会产生问题。请说明每种情况各产生哪些问题。

↓ done

↓ done

Part 03

基于Windows的编程

第 19 章

Windows平台下线程的使用

本章除了介绍 Windows 平台下创建线程的方法，还将介绍内核对象、所有者、计数器等 Windows 系统相关内容。这些内容是 Windows 编程的基础，各位若对 Windows 感兴趣就需重点学习本章。

19.1 内核对象

要想掌握Windows平台下的线程，应首先理解“内核对象”（Kernel Objects）的概念。如果仅介绍Windows平台下的线程使用技巧，则可以省略相对陌生的内核对象相关内容。但这非我所愿，而且这种方式对各位也没有帮助。

内核对象的定义

操作系统创建的资源（Resource）有很多种，如进程、线程、文件及即将介绍的信号量、互斥量等。其中大部分都是通过程序员的请求创建的，而且请求方式（请求中使用的函数）各不相同。虽然存在一些差异，但它们之间也有如下共同点：

“都是由Windows操作系统创建并管理的资源。”

不同资源类型在“管理”方式上也有差异。例如，文件管理中应注册并更新文件相关的数据I/O位置、文件的打开模式（read or write）等。如果是线程，则应注册并维护线程ID、线程所属进程等信息。操作系统为了以记录相关信息的方式管理各种资源，在其内部生成数据块（亦可视为结构体变量）。当然，每种资源需要维护的信息不同，所以每种资源拥有的数据块格式也有差异。这类数据块称为“内核对象”。

假设在Windows下创建了mydata.txt文件，此时Windows操作系统将生成1个数据块以便管理，该数据块就是内核对象。同理，Windows在创建进程、线程、线程同步信号量时也会生成相应的内核对象，用于管理操作系统资源。相信各位已经理解了内核对象的概念。

内核对象归操作系统所有

线程、文件等资源的创建请求均在进程内部完成，因此，很容易产生“此时创建的内核对象所有者就是进程”的错觉。其实，内核对象所有者是内核（操作系统）。“所有者是内核”具有如下含义：

“内核对象的创建、管理、销毁时机的决定等工作均由操作系统完成！”

内核对象就是为了管理线程、文件等资源而由操作系统创建的数据块，其创建者和所有者均为操作系统。

19.2 基于 Windows 的线程创建

如果各位学习过第18章，那应该已经掌握了线程的概念、同步的必要性等线程相关全部理论。如果之前跳过了第18章，希望大家先学习这些内容。

进程和线程的关系

既然在第18章学习过线程，那么请回答如下问题：

“程序开始运行后，调用main函数的主体是进程还是线程？”

调用main函数的主体是线程！实际上，过去的正确答案可能是进程（特别是在UNIX系列的操作系统中）。因为早期的操作系统并不支持线程，为了创建线程，经常需要特殊的库函数支持。换言之，操作系统无法意识到线程的存在，而进程实际上成为运行的最小单位。即便在这种情况下，需要线程的程序员们也会利用特殊的库函数，以拆分进程运行时间的方式创建线程。但归根结底，这仅仅是应用程序级别创建的线程，与现在讨论的操作系统级别的线程存在巨大差异。现代的Linux系列、Windows系列及各种规模不等的操作系统都在操作系统级别支持线程，因此，非显式创建线程的程序（如基于select的服务器端）可描述如下：

“单一线程模型的应用程序”

反之，显式创建单独线程的程序可描述如下：

“多线程模型的应用程序”

这就意味着main函数的运行同样基于线程完成，此时进程可以比喻为装有线程的篮子。实际的运行主体是线程。

Windows 中线程的创建方法

理解了线程的概念及工作原理后，下面介绍创建线程时使用的函数。调用该函数将创建线程，

操作系统为了管理这些资源也将同时创建内核对象。最后返回用于区分内核对象的整数型“句柄”（Handle）。第1章已介绍过，句柄相当于Linux的文件描述符。

```
#include <windows.h>

HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
);
```

➔ 成功时返回线程句柄，失败时返回 NULL。

- lpThreadAttributes　线程安全相关信息，使用默认设置时传递NULL。
- dwStackSize　要分配给线程的栈大小，传递0时生成默认大小的栈。
- lpStartAddress　传递线程的main函数信息。
- lpParameter　调用main函数时传递的参数信息。
- dwCreationFlags　用于指定线程创建后的行为，传递0时，线程创建后立即进入可执行状态。
- lpThreadId　用于保存线程ID的变量地址值。

上述定义看起来有些复杂，其实只需要考虑lpStartAddress和lpParameter这2个参数，剩下的只需传递0或NULL即可。

提　示

Windows 线程的销毁时间点

Windows 线程在首次调用的线程 main 函数返回时销毁（销毁时间点和销毁方法与 Linux 不同）。还有其他方法可以终止线程，但最好的方法就是让线程 main 函数终止（返回），故省略其他说明。

编写多线程程序的环境设置

VC++环境下需要设置“C/C++ Runtime Library”（以下简称CRT），这是调用C/C++标准函数时必需的库。过去的VC++6.0版默认只包含支持单线程的（只能在单线程模型下正常工作的）库，需要自行配置。但各位使用的VC++ Express Edition 2005或更高版本中只有支持多线程的库，不用另行设置环境。即便如此，还是通过图19-1给出CRT的指定位置。在菜单中选择“项目”→“属性”，或使用快捷键Alt+F7打开库的配置页，即可看到相关界面。

图19-1 指定CRT

从图19-1的“运行库”中可以看到，可选的4种库都与多线程有关。VC++6.0版的用户配置界面稍有区别。打开工程属性（也可以用Alt+F7）进入C/C++相关页，将“Use run-time library”区域改为“Multithread DLL”。

创建“使用线程安全标准 C 函数”的线程

之前介绍过创建线程时使用的CreateThread函数，如果线程要调用C/C++标准函数，需要通过如下方法创建线程。因为通过CreateThread函数调用创建出的线程在使用C/C++标准函数时并不稳定。

```
#include <process.h>

uintptr_t _beginthreadex(
    void * security,
    unsigned stack_size,
    unsigned (* start_address)(void *),
    void * arglist,
    unsigned initflag,
    unsigned * thrdaddr
);
```

➔ 成功时返回线程句柄，失败时返回 0。

上述函数与之前的CreateThread函数相比，参数个数及各参数的含义和顺序均相同，只是变

量名和参数类型有所不同。因此，用上述函数替换CreateThread函数时，只需适当更改数据类型。下面通过上述函数编写示例。

定义于_beginthreadex函数之前的_beginthread函数

如果查阅_beginthreadex 函数相关资料，会发现还有更好用的_beginthread函数。但该函数的问题在于，它会让创建线程时返回的句柄失效，以防止访问内核对象。_beginthreadex 就是为了解决这一问题而定义的函数。

上述函数的返回值类型uintptr_t是64位unsigned整数型。但下述示例将通过声明CreateThread函数的返回值类型HANDLE（这同样是整数型）保存返回的线程句柄。

❖ thread1_win.c

```
1.  #include <stdio.h>
2.  #include <windows.h>
3.  #include <process.h>    /* _beginthreadex, _endthreadex */
4.  unsigned WINAPI ThreadFunc(void *arg);
5.
6.  int main(int argc, char *argv[])
7.  {
8.      HANDLE hThread;
9.      unsigned threadID;
10.     int param=5;
11.
12.     hThread=(HANDLE)_beginthreadex(NULL, 0, ThreadFunc, (void*)&param, 0, &threadID);
13.     if(hThread==0)
14.     {
15.         puts("_beginthreadex() error");
16.         return -1;
17.     }
18.     Sleep(3000);
19.     puts("end of main");
20.     return 0;
21. }
22.
23. unsigned WINAPI ThreadFunc(void *arg)
24. {
25.     int i;
26.     int cnt=*((int*)arg);
27.     for(i=0; i<cnt; i++)
28.     {
29.         Sleep(1000); puts("running thread");
30.     }
31.     return 0;
32. }
```

代码说明

- 第12行：将ThreadFunc作为线程的main函数，并向其传递变量param的地址值，同时请求创建线程。
- 第18行：Sleep函数以1/1000秒为单位进入阻塞状态，因此，传入3000时将等待3秒钟。
- 第23行：WINAPI是Windows固有的关键字，它用于指定参数传递方向、分配的栈返回方式等函数调用相关规定。插入它是为了遵守_beginthreadex函数要求的调用规定。

❖ 运行结果：thread1_win.c one

```
running thread
running thread
running thread
end of main
```

❖ 运行结果：thread1_win.c two

```
running thread
running thread
end of main
running thread
```


与Linux相同，Windows同样在main函数返回后终止进程，也同时终止其中包含的所有线程。因此，需要特殊方法解决该问题。另外，从上述运行结果中可以看到，最后输出的并非字符串“end of main”，而是“running thread”。但这是在main函数返回后、完全销毁进程前输出的字符串。

知识补给站 句柄、内核对象和ID间的关系

线程也属于操作系统管理的资源，因此会伴随着内核对象的创建，并为了引用内核对象而返回句柄。可以利用句柄发送如下请求：

“我会一直等到该句柄指向的线程终止。”

可以通过句柄区分内核对象，通过内核对象可以区分线程。最终，线程句柄成为区分线程的工具。那线程ID又是什么呢？如上述示例所示，通过_beginthreadex函数的最后一个参数可以获取线程ID。各位或许对句柄和ID的并存感到困惑，其实它们有如下显著特点：

“句柄的整数值在不同进程中可能出现重复，但线程ID在跨进程范围内不会出现重复。”

线程ID用于区分操作系统创建的所有线程，但通常没有这种需求。建议各位多关注句柄和内核对象。

19.3　内核对象的 2 种状态

资源类型不同，内核对象也含有不同信息。其中，应用程序实现过程中需要特别关注的信息被赋予某种“状态”（state）。例如，线程内核对象中需要重点关注线程是否已终止，所以终止状态又称“signaled状态”，未终止状态称为“non-signaled状态”。

内核对象的状态及状态查看

我们通常比较关注进程的终止时间和线程的终止时间，所以自然会问：“该进程何时终止？”或“该线程何时终止？”操作系统将这些重要信息保存到内核对象，同时给出如下约定：

“进程或线程终止时，我会把相应的内核对象改为signaled状态！”

这也意味着，进程和线程的内核对象初始状态是non-signaled状态。那么，内核对象的signaled、non-signaled状态究竟如何表示呢？非常简单！通过1个boolean变量表示。内核对象带有1个boolean变量，其初始值为FALSE，此时的状态就是non-signaled状态。如果发生约定的情况（“发生了事件”，以下会酌情使用这种表达），把该变量改为TRUE，此时的状态就是signaled状态。内核对象类型不同，进入signaled状态的情况也有所区别。关于这些细节，必要时再逐一介绍。

正常运行thread1_win.c示例前需要考虑如下问题：

“该内核对象当前是否为signaled状态？”

为回答类似问题，系统定义了WaitForSingleObject和WaitForMultipleObjects函数。

WaitForSingleObject & WaitForMultipleObjects

首先介绍WaitForSingleObject函数，该函数针对单个内核对象验证signaled状态。

```
#include <windows.h>

DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);
```

➔ 成功时返回事件信息，失败时返回 WAIT_FAILED。

hHandle	查看状态的内核对象句柄。
dwMilliseconds	以1/1000秒为单位指定超时，传递INFINITE时函数不会返回，直到内核对象变成signaled状态。
返回值	进入signaled状态返回WAIT_OBJECT_0，超时返回WAIT_TIMEOUT。

该函数由于发生事件（变为signaled状态）返回时，有时会把相应内核对象再次改为

non-signaled状态。这种可以再次进入non-signaled状态的内核对象称为“auto-reset模式”的内核对象，而不会自动跳转到non-signaled状态的内核对象称为“manual-reset模式”的内核对象。即将介绍的函数与上述函数不同，可以验证多个内核对象状态。

```
#include <windows.h>

DWORD WaitForMultipleObjects(
    DWORD nCount, const HANDLE * lpHandles, BOOL bWaitAll, DWORD dwMilliseconds);
```

➔ 成功时返回事件信息，失败时返回 WAIT_FAILED。

- nCount：需验证的内核对象数。
- lpHandles：存有内核对象句柄的数组地址值。
- bWaitAll：如果为TRUE，则所有内核对象全部变为signaled时返回；如果为FALSE，则只要有1个验证对象的状态变为signaled就会返回。
- dwMilliseconds：以1/1000秒为单位指定超时，传递INFINITE时函数不会返回，直到内核对象变为signaled状态。

下面利用WaitForSingleObject函数尝试解决示例thread1_win.c的问题。

❖ thread2_win.c

```
1.  #include <stdio.h>
2.  #include <windows.h>
3.  #include <process.h>    /* _beginthreadex, _endthreadex */
4.  unsigned WINAPI ThreadFunc(void *arg);
5.
6.  int main(int argc, char *argv[])
7.  {
8.      HANDLE hThread;
9.      DWORD wr;
10.     unsigned threadID;
11.     int param=5;
12.
13.     hThread=(HANDLE)_beginthreadex(NULL, 0, ThreadFunc, (void*)&param, 0, &threadID);
14.     if(hThread==0)
15.     {
16.         puts("_beginthreadex() error");
17.         return -1;
18.     }
19.
20.     if((wr=WaitForSingleObject(hThread, INFINITE))==WAIT_FAILED)
21.     {
22.         puts("thread wait error");
23.         return -1;
24.     }
25.
26.     printf("wait result: %s \n", (wr==WAIT_OBJECT_0) ? "signaled":"time-out");
```

```
27.     puts("end of main");
28.     return 0;
29. }
30.
31. unsigned WINAPI ThreadFunc(void *arg)
32. {
33.     int i;
34.     int cnt=*((int*)arg);
35.     for(i=0; i<cnt; i++)
36.     {
37.         Sleep(1000); puts("running thread");
38.     }
39.     return 0;
40. }
```

代码说明

- 第20行：调用WaitForSingleObject函数等待线程终止。
- 第26行：利用WaitForSingleObject函数的返回值查看返回原因。

❖ 运行结果：thread2_win.c

```
running thread
running thread
running thread
running thread
running thread
wait result: signaled
end of main
```

运行结果显示，thread1_win.c中存在的问题得到解决。下列示例将涉及WaitForMultipleObjects函数的应用。但该函数与WaitForSingleObject相比没有太大区别，使用方法也相对简单。

WaitForSingleObject & WaitForMultipleObjects

第18章在Linux平台下分析了临界区问题，本章最后的内容将留给Windows平台下的临界区问题。该示例只是第18章thread4.c示例的Windows版（创建的线程数减少为50个），不再另做说明。

❖ thread3_win.c

```
1.  #include <stdio.h>
2.  #include <windows.h>
3.  #include <process.h>
4.
5.  #define NUM_THREAD  50
6.  unsigned WINAPI threadInc(void * arg);
7.  unsigned WINAPI threadDes(void * arg);
```

```
8.  long long num=0;
9.
10. int main(int argc, char *argv[])
11. {
12.     HANDLE tHandles[NUM_THREAD];
13.     int i;
14.
15.     printf("sizeof long long: %d \n", sizeof(long long));
16.     for(i=0; i<NUM_THREAD; i++)
17.     {
18.         if(i%2)
19.             tHandles[i]=(HANDLE)_beginthreadex(NULL, 0, threadInc, NULL, 0, NULL);
20.         else
21.             tHandles[i]=(HANDLE)_beginthreadex(NULL, 0, threadDes, NULL, 0, NULL);
22.     }
23.
24.     WaitForMultipleObjects(NUM_THREAD, tHandles, TRUE, INFINITE);
25.     printf("result: %lld \n", num);
26.     return 0;
27. }
28.
29. unsigned WINAPI threadInc(void * arg)
30. {
31.     int i;
32.     for(i=0; i<50000000; i++)
33.         num+=1;
34.     return 0;
35. }
36. unsigned WINAPI threadDes(void * arg)
37. {
38.     int i;
39.     for(i=0; i<50000000; i++)
40.         num-=1;
41.     return 0;
42. }
```


❖ 运行结果：thread3_win.c

```
sizeof long long: 8
result: 71007761
```

即使多运行几次也无法得到正确结果，而且每次结果都不同。可以利用第20章的同步技术得到预想的结果。

19.4 习题

(1) 下列关于内核对象的说法错误的是?

a. 内核对象是操作系统保存各种资源信息的数据块。

b. 内核对象的所有者是创建该内核对象的进程。

c. 由用户进程创建并管理内核对象。

d. 无论操作系统创建和管理的资源类型是什么，内核对象的数据块结构都完全相同。

(2) 现代操作系统大部分都在操作系统级别支持线程。根据该情况判断下列描述中错误的是？

a. 调用main函数的也是线程。

b. 如果进程不创建线程，则进程内不存在任何线程。

c. 多线程模型是进程内可以创建额外线程的程序类型。

d. 单一线程模型是进程内只额外创建1个线程的程序模型。

(3) 请比较从内存中完全销毁Windows线程和Linux线程的方法。

(4) 通过线程创建过程解释内核对象、线程、句柄之间的关系。

(5) 判断下列关于内核对象描述的正误。

- ❑ 内核对象只有signaled和non-signaled这2种状态。(　　)
- ❑ 内核对象需要转为signaled状态时，需要程序员亲自将内核对象的状态改为signaled状态。(　　)
- ❑ 线程的内核对象在线程运行时处于signaled状态，线程终止则进入non-signaled状态。(　　)

(6) 请解释“auto-reset模式”和“manual-reset模式”的内核对象。区分二者的内核对象特征是什么？

done!