“黑色经典”系列之《嵌入式 Linux 应用程序开发详解》

第 9 章 多线程编程

本章目标

在前两章中，读者主要学习了有关进程控制和进程间通信的开发，这些都是 Linux 中开发的基础。在这一章中将学习轻量级进程——线程的开发，由于线程的高效性和可操作性，在大型程序开发中运用得非常广泛，希望读者能够很好地掌握。

- 掌握 Linux 中线程的基本概念
- 掌握 Linux 中线程的创建及使用
- 掌握 Linux 中线程属性的设置
- 能够独立编写多线程程序
- 能够处理多线程中的变量问题
- 能够处理多线程中的同步文件

9.1 Linux 下线程概述

9.1.1 线程概述

前面已经提到，进程是系统中程序执行和资源分配的基本单位。每个进程都拥有自己的数据段、代码段和堆栈段，这就造成了进程在进行切换等操作时都需要有比较负责的上下文切换等动作。为了进一步减少处理机的空转时间支持多处理器和减少上下文切换开销，进程在演化中出现了另一个概念——线程。它是一个进程内的基本调度单位，也可以称为轻量级进程。线程是在共享内存空间中并发的多道执行路径，它们共享一个进程的资源，如文件描述和信号处理。因此，大大减少了上下文切换的开销。

同进程一样，线程也将相关的变量值放在线程控制表内。一个进程可以有多个线程，也就是有多个线程控制表及堆栈寄存器，但却共享一个用户地址空间。要注意的是，由于线程共享了进程的资源和地址空间，因此，任何线程对系统资源的操作都会给其他线程带来影响，因此，多线程中的同步就是非常重要的问题了。在多线程系统中，进程与进程的关系如表 9.1 所示。

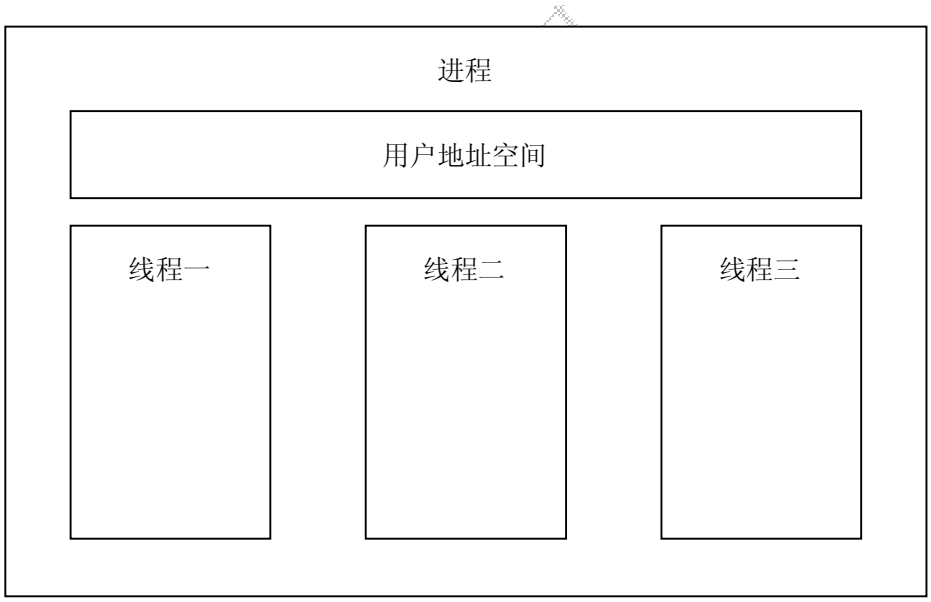

图 9.1 进程与线程关系

9.1.2 线程分类

线程按照其调度者可以分为用户级线程和核心级线程两种。

（1）用户级线程

用户级线程主要解决的是上下文切换的问题，它的调度算法和调度过程全部由用户自行选择决定，在运行时不需要特定的内核支持。在这里，操作系统往往会提供一个用户空间的线程库，该线程库提供了线程的创建、调度、撤销等功能，而内核仍然仅对进程进行管理。

如果一个进程中的某一个线程调用了一个阻塞的系统调用，那么该进程包括该进程中的其他所有线程也同时被阻塞。这种用户级线程的主要缺点是在一个进程中的多个线程的调度中无法发挥多处理器的优势。

（2）核心级线程

这种线程允许不同进程中的线程按照同一相对优先调度方法进行调度，这样就可以发挥多处理器的并发优势。

现在大多数系统都采用用户级线程与核心级线程并存的方法。一个用户级线程可以对应一个或几个核心级线程，也就是“一对一”或“多对一”模型。这样既可满足多处理机系统的需要，也可以最大限度地减少调度开销。

9.1.3 Linux 线程技术的发展

在 Linux 中，线程技术也经过了一代代的发展过程。

在 Linux2.2 内核中，并不存在真正意义上的线程。当时 Linux 中常用的线程 pthread 实际上是通过进程来模拟的，也就是说 Linux 中的线程也是通过 fork 创建的“轻”进程，并且线程的个数也很有限，最多只能有 4096 个进程/线程同时运行。

Linux2.4 内核消除了这个线程个数的限制，并且允许在系统运行中动态地调整进程数上限。当时采用的是 LinuxThread 线程库，它对应的线程模型是“一对一”线程模型，也就是一个用户级线程对应一个内核线程，而线程之间的管理在内核外函数库中实现。这种线程模型得到了广泛应用。但是，LinuxThread 也由于 Linux 内核的限制以及实现难度等原因，并不是完全与 POSIX 兼容。另外，它的进程 ID、信号处理、线程总数、同步等各方面都还有诸多的问题。

为了解决以上问题，在 Linux2.6 内核中，进程调度通过重新编写，删除了以前版本中效率不高的算法。内核线程框架也被重新编写，开始使用 NPTL（Native POSIX Thread Library）线程库。这个线程库有以下几点设计目标：POSIX 兼容性、多处理器结构的应用、低启动开销、低链接开销、与 LinuxThreads 应用的二进制兼容性、软硬件的可扩展能力、与 C++集成等。这一切都使得 Linux2.6 内核的线程机制更加完备，能够更好地完成其设计目标。与 LinuxThreads 不同，NPTL 没有使用管理线程，核心线程的管理直接放在核内进行，这也带了性能的优化。由于 NPTL 仍然采用 1：1 的线程模型，NPTL 仍然不是 POSIX 完全兼容的，但就性能而言相对 LinuxThreads 已经有很大程度上的改进。

9.2 Linux 线程实现

9.2.1 线程基本操作

这里要讲的线程相关操作都是用户空间线程的操作。在 Linux 中，一般 Pthread 线程库是一套通用的线程库，是由 POSIX 提出的，因此具有很好的可移植性。

1．线程创建和退出

（1）函数说明

创建线程实际上就是确定调用该线程函数的入口点，这里通常使用的函数是 pthread_create。在线程创建以后，就开始运行相关的线程函数，在该函数运行完之后，该线程也就退出了，这也是线程退出一种方法。另一种退出线程的方法是使用函数 pthread_exit，这是线程的主动行为。这里要注意的是，在使用线程函数时，不能随意使用 exit 退出函数进行出错处理，由于 exit 的作用是使调用进程终止，往往一个进程包含多个线程，因此，在使用 exit 之后，该进程中的所有线程都终止了。因此，在线程中就可以使用 pthread_exit 来代替进程中的 exit。

由于一个进程中的多个线程是共享数据段的，因此通常在线程退出之后，退出线程所占用的资源并不会随着线程的终止而得到释放。正如进程之间可以用 wait()系统调用来同步终止并释放资源一样，线程之间也有类似机制，那就是 pthread_join()函数。pthread_join 可以用于将当前线程挂起，等待线程的结束。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源就被收回。

（2）函数格式

表 9.1 列出了 pthread_create 函数的语法要点。

表 9.1 pthread_create 函数语法要点

所需头文件	#include <pthread.h>
函数原型	int pthread_create ((pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg))
函数传入值	thread：线程标识符
	attr：线程属性设置（具体设定在 9.2.2 会进行讲解）
	续表
	start_routine：线程函数的起始地址
	arg：传递给 start_routine 的参数
函数返回值	成功：0
	出错：−1

表 9.2 列出了 pthread_exit 函数的语法要点。

表 9.2 pthread_exit 函数语法要点

所需头文件	#include <pthread.h>
函数原型	void pthread_exit(void *retval)
函数传入值	Retval：pthread_exit()调用者线程的返回值，可由其他函数如 pthread_join 来检索获取

表 9.3 列出了 pthread_join 函数的语法要点。

表 9.3 pthread_join 函数语法要点

所需头文件	#include <pthread.h>
函数原型	int pthread_join ((pthread_t th, void **thread_return))

函数传入值	th：等待线程的标识符
	thread_return：用户定义的指针，用来存储被等待线程的返回值（不为 NULL 时）
函数返回值	成功：0
	出错：−1

（3）函数使用

以下实例中创建了两个线程，其中第一个线程是在程序运行到中途时调用 pthread_exit 函数退出，第二个线程正常运行退出。在主线程中收集这两个线程的退出信息，并释放资源。从这个实例中可以看出，这两个线程是并发运行的。

```
/*thread.c*/
#include <stdio.h>
#include <pthread.h>
/*线程一*/
void thread1(void)
{
     int i=0;
     for(i=0;i<6;i++){
          printf("This is a pthread1.\n");
          if(i==2)
               pthread_exit(0);
          sleep(1);
          }
}
/*线程二*/
void thread2(void)
{
     int i;
     for(i=0;i<3;i++)
          printf("This is a pthread2.\n");
     pthread_exit(0);
}

int main(void)
{
     pthread_t id1,id2;
     int i,ret;
/*创建线程一*/
    ret=pthread_create(&id1,NULL,(void *) thread1,NULL);
```

```
        if(ret!=0){
                printf ("Create pthread error!\n");
        exit (1);
        }
    /*创建线程二*/
        ret=pthread_create(&id2,NULL,(void *) thread2,NULL);
        if(ret!=0){
                printf ("Create pthread error!\n");
                exit (1);
        }
    /*等待线程结束*/
        pthread_join(id1,NULL);
        pthread_join(id2,NULL);
        exit (0);
    }
```

以下是程序运行结果：

```
 [root@(none) tmp]# ./thread
This is a pthread1.
This is a pthread2.
This is a pthread2.
This is a pthread2.
This is a pthread1.
This is a pthread1.
```

2．修改线程属性

（1）函数说明

读者是否还记得 pthread_create 函数的第二个参数——线程的属性。在上一个实例中，将该值设为 NULL，也就是采用默认属性，线程的多项属性都是可以更改的。这些属性主要包括绑定属性、分离属性、堆栈地址、堆栈大小、优先级。其中系统默认的属性为非绑定、非分离、缺省 1M 的堆栈、与父进程同样级别的优先级。下面首先对绑定属性和分离属性的基本概念进行讲解。

- 绑定属性

前面已经提到，Linux 中采用“一对一”的线程机制，也就是一个用户线程对应一个内核线程。绑定属性就是指一个用户线程固定地分配给一个内核线程，因为 CPU 时间片的调度是面向内核线程（也就是轻量级进程）的，因此具有绑定属性的线程可以保证在需要的时候总有一个内核线程与之对应。而与之相对的非绑定属性就是指用户线程和内核线程的关系不是始终固定的，而是由系统来控制分配的。

- 分离属性

分离属性是用来决定一个线程以什么样的方式来终止自己。在非分离情况下，当一个线

程结束时，它所占用的系统资源并没有被释放，也就是没有真正的终止。只有当 pthread_join() 函数返回时，创建的线程才能释放自己占用的系统资源。而在分离属性情况下，一个线程结束时立即释放它所占有的系统资源。这里要注意的一点是，如果设置一个线程的分离属性，而这个线程运行又非常快，那么它很可能在 pthread_create 函数返回之前就终止了，它终止以后就可能将线程号和系统资源移交给其他的线程使用，这时调用 pthread_create 的线程就得到了错误的线程号。

这些属性的设置都是通过一定的函数来完成的，通常首先调用 pthread_attr_init 函数进行初始化，之后再调用相应的属性设置函数。设置绑定属性的函数为 pthread_attr_setscope，设置线程分离属性的函数为 pthread_attr_setdetachstate，设置线程优先级的相关函数为 pthread_attr_getschedparam（获取线程优先级）和 pthread_attr_setschedparam（设置线程优先级）。在设置完这些属性后，就可以调用 pthread_create 函数来创建线程了。

（2）函数格式

表 9.4 列出了 pthread_attr_init 函数的语法要点。

表 9.4　　pthread_attr_init 函数语法要点

所需头文件	#include <pthread.h>
函数原型	int pthread_attr_init(pthread_attr_t *attr)
函数传入值	attr：线程属性
函数返回值	成功：0
	出错：−1

表 9.5 列出了 pthread_attr_setscope 函数的语法要点。

表 9.5　　pthread_attr_setscope 函数语法要点

所需头文件	#include <pthread.h>	
函数原型	int pthread_attr_setscope(pthread_attr_t *attr, int scope)	
函数传入值	attr：线程属性	
	scope	PTHREAD_SCOPE_SYSTEM：绑定
		PTHREAD_SCOPE_PROCESS：非绑定
函数返回值	成功：0	
	出错：−1	

表 9.6 列出了 pthread_attr_setdetachstate 函数的语法要点。

表 9.6　　pthread_attr_setdetachstate 函数语法要点

所需头文件	#include <pthread.h>	
函数原型	int pthread_attr_setscope(pthread_attr_t *attr, int detachstate)	
函数传入值	attr：线程属性	
	detachstate	PTHREAD_CREATE_DETACHED：分离

		PTHREAD _CREATE_JOINABLE：非分离
函数返回值	成功：0	
	出错：–1	

表 9.7 列出了 pthread_attr_getschedparam 函数的语法要点。

表 9.7 pthread_attr_getschedparam 函数语法要点

所需头文件	#include <pthread.h>
函数原型	int pthread_attr_getschedparam (pthread_attr_t *attr, struct sched_param *param)
函数传入值	attr：线程属性
	param：线程优先级
函数返回值	成功：0
	出错：–1

表 9.8 列出了 pthread_attr_setschedparam 函数的语法要点。

表 9.8 pthread_attr_setschedparam 函数语法要点

所需头文件	#include <pthread.h>
函数原型	int pthread_attr_setschedparam (pthread_attr_t *attr, struct sched_param *param)
函数传入值	attr：线程属性
	param：线程优先级
函数返回值	成功：0
	出错：–1

3．使用实例

该实例将上一节中的第一个线程设置为分离属性，并将第二个线程设置为始终运行状态，这样就可以在第二个线程运行过程中查看内存值的变化。

其源代码如下所示：

```
/*pthread.c*/
#include <stdio.h>
#include <pthread.h>
#include <time.h>
/*线程一*/
void thread1(void)
{
      int i=0;
      for(i=0;i<6;i++){
            printf("This is a pthread1.\n");
            if(i==2)
```

```
                pthread_exit(0);
            sleep(1);
        }
}
/*线程二*/
void thread2(void)
{
        int i;
        while(1){
        for(i=0;i<3;i++)
            printf("This is a pthread2.\n");
        sleep(1);}
        pthread_exit(0);
}

int main(void)
{
        pthread_t id1,id2;
        int i,ret;
        pthread_attr_t attr;
/*初始化线程*/
        pthread_attr_init(&attr);
/*设置线程绑定属性*/
        pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
/*设置线程分离属性*/
        pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
/*创建线程*/
        ret=pthread_create(&id1,&attr,(void *) thread1,NULL);
        if(ret!=0){
            printf ("Create pthread error!\n");
        exit (1);
        }
        ret=pthread_create(&id2,NULL,(void *) thread2,NULL);
        if(ret!=0){
            printf ("Create pthread error!\n");
            exit (1);
        }
        pthread_join(id2,NULL);
        return (0);
```

```
}
```

接下来可以在线程一运行前后使用“free”命令查看内存使用情况。以下是运行结果：

```
[root@(none) tmp]# ./thread3
This is a pthread1.
This is a pthread2.
This is a pthread2.
This is a pthread2.
This is a pthread1.
This is a pthread2.
This is a pthread2.
This is a pthread2.
This is a pthread1.
This is a pthread2.
…
[root@www yul]# free
             total       used       free     shared    buffers     cached
Mem:         1028428     570212     458216         48      204292      93196
-/+ buffers/cache:  272724     755704
Swap:       1020116          0     1020116
[root@www yul]# free
             total       used       free     shared    buffers     cached
Mem:         1028428     570220     458208         48     204296      93196
-/+ buffers/cache:     272728     755700
Swap:        1020116          0    1020116
[root@www yul]# free
             total       used       free     shared    buffers     cached
Mem:         1028428     570212     458216         48     204296      93196
-/+ buffers/cache:    272720     755708
Swap:       1020116          0    1020116
```

可以看到，线程一在运行结束后就收回了系统资源，释放了内存。

9.2.2 线程访问控制

由于线程共享进程的资源和地址空间，因此在对这些资源进行操作时，必须考虑到线程间资源访问的惟一性问题，这里主要介绍 POSIX 中线程同步的方法，主要有互斥锁和信号量的方式。

1．mutex 互斥锁线程控制

（1）函数说明

mutex 是一种简单的加锁的方法来控制对共享资源的存取。这个互斥锁只有两种状态，也就是上锁和解锁，可以把互斥锁看作某种意义上的全局变量。在同一时刻只能有一个线程掌握某个互斥上的锁，拥有上锁状态的线程能够对共享资源进行操作。若其他线程希望上锁一个已经上锁了的互斥锁，则该线程就会挂起，直到上锁的线程释放掉互斥锁为止。可以说，这把互斥锁使得共享资源按序在各个线程中操作。

互斥锁的操作主要包括以下几个步骤。

- 互斥锁初始化：pthread_mutex_init
- 互斥锁上锁：pthread_mutex_lock
- 互斥锁判断上锁：pthread_mutex_trylock
- 互斥锁接锁：pthread_mutex_unlock
- 消除互斥锁：pthread_mutex_destroy

其中，互斥锁可以分为快速互斥锁、递归互斥锁和检错互斥锁。这三种锁的区别主要在于其他未占有互斥锁的线程在希望得到互斥锁时的是否需要阻塞等待。快速锁是指调用线程会阻塞直至拥有互斥锁的线程解锁为止。递归互斥锁能够成功地返回并且增加调用线程在互斥上加锁的次数，而检错互斥锁则为快速互斥锁的非阻塞版本，它会立即返回并返回一个错误信息。

（2）函数格式

表 9.9 列出了 pthread_mutex_init 函数的语法要点。

表 9.9　　pthread_mutex_init 函数语法要点

所需头文件	#include <pthread.h>	
函数原型	int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)	
函数传入值	Mutex：互斥锁	

续表

函数传入值	Mutexattr	PTHREAD_MUTEX_INITIALIZER：创建快速互斥锁
		PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP：创建递归互斥锁
		PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP：创建检错互斥锁
函数返回值	成功：0	
	出错：−1	

表 9.10 列出了 pthread_mutex_lock 等函数的语法要点。

表 9.10　　pthread_mutex_lock 等函数语法要点

所需头文件	#include <pthread.h>
函数原型	int pthread_mutex_lock(pthread_mutex_t *mutex,) int pthread_mutex_trylock(pthread_mutex_t *mutex,)

	int pthread_mutex_unlock(pthread_mutex_t *mutex,) int pthread_mutex_destroy(pthread_mutex_t *mutex,)
函数传入值	Mutex：互斥锁
函数返回值	成功：0
	出错：−1

（3）使用实例

该实例使用互斥锁来实现对变量 lock_var 的加一、打印操作。

```
/*mutex.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int lock_var;
time_t end_time;

void pthread1(void *arg);
void pthread2(void *arg);

int main(int argc, char *argv[])
{
     pthread_t id1,id2;
     pthread_t mon_th_id;
     int ret;

     end_time = time(NULL)+10;
/*互斥锁初始化*/
     pthread_mutex_init(&mutex,NULL);
/*创建两个线程*/
     ret=pthread_create(&id1,NULL,(void *)pthread1, NULL);
     if(ret!=0)
           perror("pthread cread1");
     ret=pthread_create(&id2,NULL,(void *)pthread2, NULL);
     if(ret!=0)
           perror("pthread cread2");
```

```
        pthread_join(id1,NULL);
        pthread_join(id2,NULL);
        exit(0);
}

void pthread1(void *arg)
{
        int i;
        while(time(NULL) < end_time){
/*互斥锁上锁*/
            if(pthread_mutex_lock(&mutex)!=0){
            perror("pthread_mutex_lock");
            }
            else
                   printf("pthread1:pthread1 lock the variable\n");
            for(i=0;i<2;i++){
                  sleep(1);
                  lock_var++;
            }
/*互斥锁接锁*/
            if(pthread_mutex_unlock(&mutex)!=0){
                  perror("pthread_mutex_unlock");
            }
            else
                  printf("pthread1:pthread1 unlock the variable\n");
        sleep(1);
        }
}

void pthread2(void *arg)
{
        int nolock=0;
        int ret;
        while(time(NULL) < end_time){
/*测试互斥锁*/
            ret=pthread_mutex_trylock(&mutex);
            if(ret==EBUSY)
                  printf("pthread2:the variable is locked by pthread1\n");
            else{
```

```
                if(ret!=0){
                        perror("pthread_mutex_trylock");
                        exit(1);
                }
                else
                        printf("pthread2:pthread2 got lock.The variable is
%d\n",lock_var);
    /*互斥锁接锁*/
                if(pthread_mutex_unlock(&mutex)!=0){
                      perror("pthread_mutex_unlock");
                }
                else
                      printf("pthread2:pthread2 unlock the variable\n");
            }
            sleep(3);
         }
    }
```

该实例的运行结果如下所示：

```
 [root@(none) tmp]# ./mutex2
pthread1:pthread1 lock the variable
pthread2:the variable is locked by pthread1
pthread1:pthread1 unlock the variable
pthread:pthread2 got lock.The variable is 2
pthread2:pthread2 unlock the variable
pthread1:pthread1 lock the variable
pthread1:pthread1 unlock the variable
pthread:pthread2 got lock.The variable is 4
pthread2:pthread2 unlock the variable
pthread1:pthread1 lock the variable
…
```

2．信号量线程控制

（1）信号量说明

在第 8 章中已经讲到，信号量也就是操作系统中所用到的 PV 原语，它广泛用于进程或线程间的同步与互斥。信号量本质上是一个非负的整数计数器，它被用来控制对公共资源的访问。这里先来简单复习一下 PV 原语的工作原理。

PV 原语是对整数计数器信号量 sem 的操作。一次 P 操作使 sem 减一，而一次 V 操作使

sem 加一。进程（或线程）根据信号量的值来判断是否对公共资源具有访问权限。当信号量 sem 的值大于等于零时，该进程（或线程）具有公共资源的访问权限；相反，当信号量 sem 的值小于零时，该进程（或线程）就将阻塞直到信号量 sem 的值大于等于 0 为止。

PV 原语主要用于进程或线程间的同步和互斥这两种典型情况。若用于互斥，几个进程（或线程）往往只设置一个信号量 sem，它们的操作流程如图 9.2 所示。

当信号量用于同步操作时，往往会设置多个信号量，并安排不同的初始值来实现它们之间的顺序执行，它们的操作流程如图 9.3 所示。

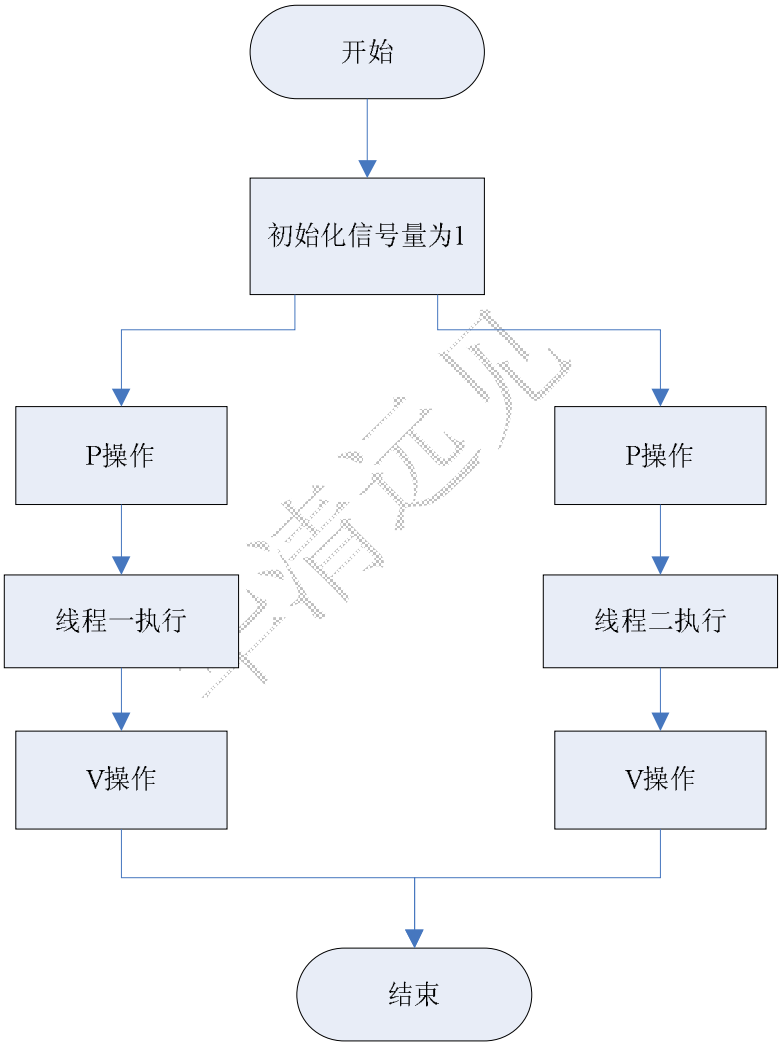

图 9.2　信号量互斥操作

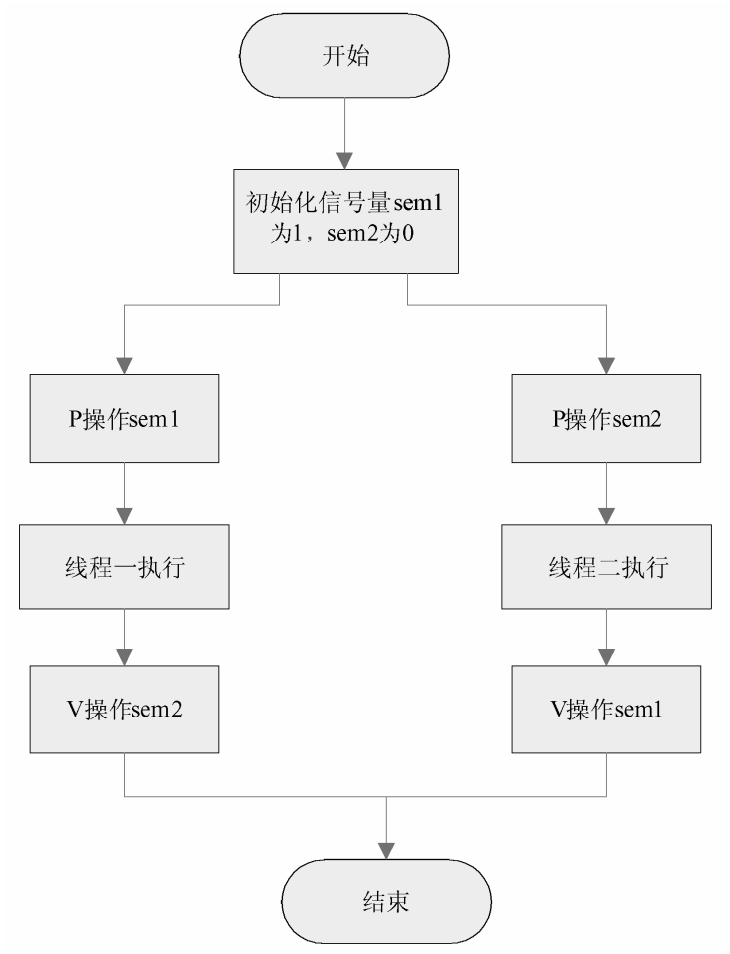

图 9.3　信号量同步操作

（2）函数说明

Linux 实现了 POSIX 的无名信号量，主要用于线程间的互斥同步。这里主要介绍几个常见函数。

- sem_init 用于创建一个信号量，并能初始化它的值。
- sem_wait 和 sem_trywait 相当于 P 操作，它们都能将信号量的值减一，两者的区别在于若信号量小于零时，sem_wait 将会阻塞进程，而 sem_trywait 则会立即返回。
- sem_post 相当于 V 操作，它将信号量的值加一同时发出信号唤醒等待的进程。
- sem_getvalue 用于得到信号量的值。
- sem_destroy 用于删除信号量。

（3）函数格式

表 9.11 列出了 sem_init 函数的语法要点。

表 9.11　　sem_init 函数语法要点

所需头文件	#include <semaphore.h>

函数原型	int sem_init(sem_t *sem,int pshared,unsigned int value)
函数传入值	sem：信号量
	pshared：决定信号量能否在几个进程间共享。由于目前 Linux 还没有实现进程间共享信号量，所以这个值只能够取 0
	value：信号量初始化值
函数返回值	成功：0
	出错：−1

表 9.12 列出了 sem_wait 等函数的语法要点。

表 9.12　　sem_wait 等函数语法要点

所需头文件	#include <pthread.h>
函数原型	int sem_wait(sem_t *sem) int sem_trywait(sem_t *sem) int sem_post(sem_t *sem) int sem_getvalue(sem_t *sem) int sem_destroy(sem_t *sem)
函数传入值	sem：信号量
函数返回值	成功：0
	出错：−1

（4）使用实例

下面实例 1 使用信号量实现了上一实例中对 lock_var 的操作，在这里使用的是互斥操作，也就是只使用一个信号量来实现。代码如下所示：

```
/*sem_mutex.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>
#include <sys/ipc.h>
#include <semaphore.h>
int lock_var;
time_t end_time;
sem_t sem;

void pthread1(void *arg);
void pthread2(void *arg);
```

```
int main(int argc, char *argv[])
{
     pthread_t id1,id2;
     pthread_t mon_th_id;
     int ret;
     end_time = time(NULL)+30;
/*初始化信号量为 1*/
     ret=sem_init(&sem,0,1);
     if(ret!=0)
     {
           perror("sem_init");
     }
/*创建两个线程*/
     ret=pthread_create(&id1,NULL,(void *)pthread1, NULL);
     if(ret!=0)
           perror("pthread cread1");
     ret=pthread_create(&id2,NULL,(void *)pthread2, NULL);
     if(ret!=0)
           perror("pthread cread2");
           pthread_join(id1,NULL);
           pthread_join(id2,NULL);

     exit(0);
}

void pthread1(void *arg)
{
     int i;
     while(time(NULL) < end_time){
/*信号量减一，P 操作*/
         sem_wait(&sem);
         for(i=0;i<2;i++){
               sleep(1);
               lock_var++;
               printf("lock_var=%d\n",lock_var);
         }
         printf("pthread1:lock_var=%d\n",lock_var);
/*信号量加一，V 操作*/
         sem_post(&sem);
```

```
            sleep(1);
        }
    }

    void pthread2(void *arg)
    {
        int nolock=0;
        int ret;
        while(time(NULL) < end_time){
    /*信号量减一，P 操作*/
            sem_wait(&sem);
            printf("pthread2:pthread1 got lock;lock_var=%d\n",lock_var);
    /*信号量加一，V 操作*/
            sem_post(&sem);
            sleep(3);
        }
    }
```

程序运行结果如下所示：

```
[root@(none) tmp]# ./sem_num
lock_var=1
lock_var=2
pthread1:lock_var=2
pthread2:pthread1 got lock;lock_var=2
lock_var=3
lock_var=4
pthread1:lock_var=4
pthread2:pthread1 got lock;lock_var=4
```

接下来是通过两个信号量来实现两个线程间的同步，仍然完成了以上实例中对 lock_var 的操作。代码如下所示：

```
/*sem_syn.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>
#include <sys/ipc.h>
#include <semaphore.h>
```

```
int lock_var;
time_t end_time;
sem_t sem1,sem2;

void pthread1(void *arg);
void pthread2(void *arg);

int main(int argc, char *argv[])
{
     pthread_t id1,id2;
     pthread_t mon_th_id;
     int ret;
     end_time = time(NULL)+30;
/*初始化两个信号量，一个信号量为 1，一个信号量为 0*/
     ret=sem_init(&sem1,0,1);
     ret=sem_init(&sem2,0,0);
     if(ret!=0)
     {
            perror("sem_init");
     }
/*创建两个线程*/
     ret=pthread_create(&id1,NULL,(void *)pthread1, NULL);
     if(ret!=0)
            perror("pthread cread1");
     ret=pthread_create(&id2,NULL,(void *)pthread2, NULL);
     if(ret!=0)
            perror("pthread cread2");
     pthread_join(id1,NULL);
     pthread_join(id2,NULL);
     exit(0);
}

void pthread1(void *arg)
{
     int i;
     while(time(NULL) < end_time){
/*P 操作信号量 2*/
            sem_wait(&sem2);
```

```
            for(i=0;i<2;i++){
                  sleep(1);
                  lock_var++;
                  printf("lock_var=%d\n",lock_var);
            }
            printf("pthread1:lock_var=%d\n",lock_var);
/*V 操作信号量 1*/
            sem_post(&sem1);
            sleep(1);
      }
}

void pthread2(void *arg)
{
      int nolock=0;
      int ret;
      while(time(NULL) < end_time){
/*P 操作信号量 1*/
      sem_wait(&sem1);
      printf("pthread2:pthread1 got lock;lock_var=%d\n",lock_var);
/*V 操作信号量 2*/
      sem_post(&sem2);
      sleep(3);
      }
}
```

从以下结果中可以看出，该程序确实实现了先运行线程二，再运行线程一。

```
[root@(none) tmp]# ./sem_num
pthread2:pthread1 got lock;lock_var=0
lock_var=1
lock_var=2
pthread1:lock_var=2
pthread2:pthread1 got lock;lock_var=2
lock_var=3
lock_var=4
pthread1:lock_var=4
```

9.3 实验内容——“生产者消费者”实验

1．实验目的

“生产者消费者”问题是一个著名的同时性编程问题的集合。通过编写经典的“生产者消费者”问题的实验，读者可以进一步熟悉 Linux 中多线程编程，并且掌握用信号量处理线程间的同步互斥问题。

2．实验内容

“生产者消费者”问题描述如下。

有一个有限缓冲区和两个线程：生产者和消费者。他们分别把产品放入缓冲区和从缓冲区中拿走产品。当一个生产者在缓冲区满时必须等待，当一个消费者在缓冲区空时页必须等待。它们之间的关系如下图所示：

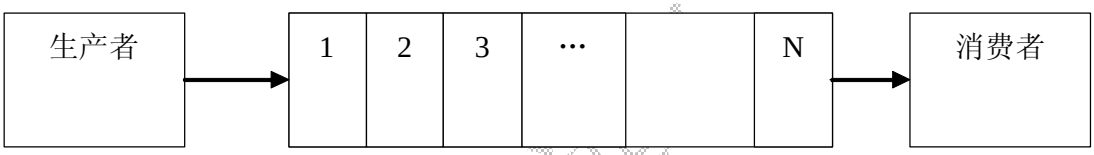

图 9.4 生产者消费者问题描述

这里要求用有名管道来模拟有限缓冲区，用信号量来解决生产者消费者问题中的同步和互斥问题。

3．实验步骤

（1）信号量的考虑

这里使用 3 个信号量，其中两个信号量 avail 和 full 分别用于解决生产者和消费者线程之间的同步问题，mutex 是用于这两个线程之间的互斥问题。其中 avail 初始化为 N（有界缓冲区的空单元数），mutex 初始化为 1，full 初始化为 0。

（2）画出流程图

本实验流程图如下图 9.5 所示。

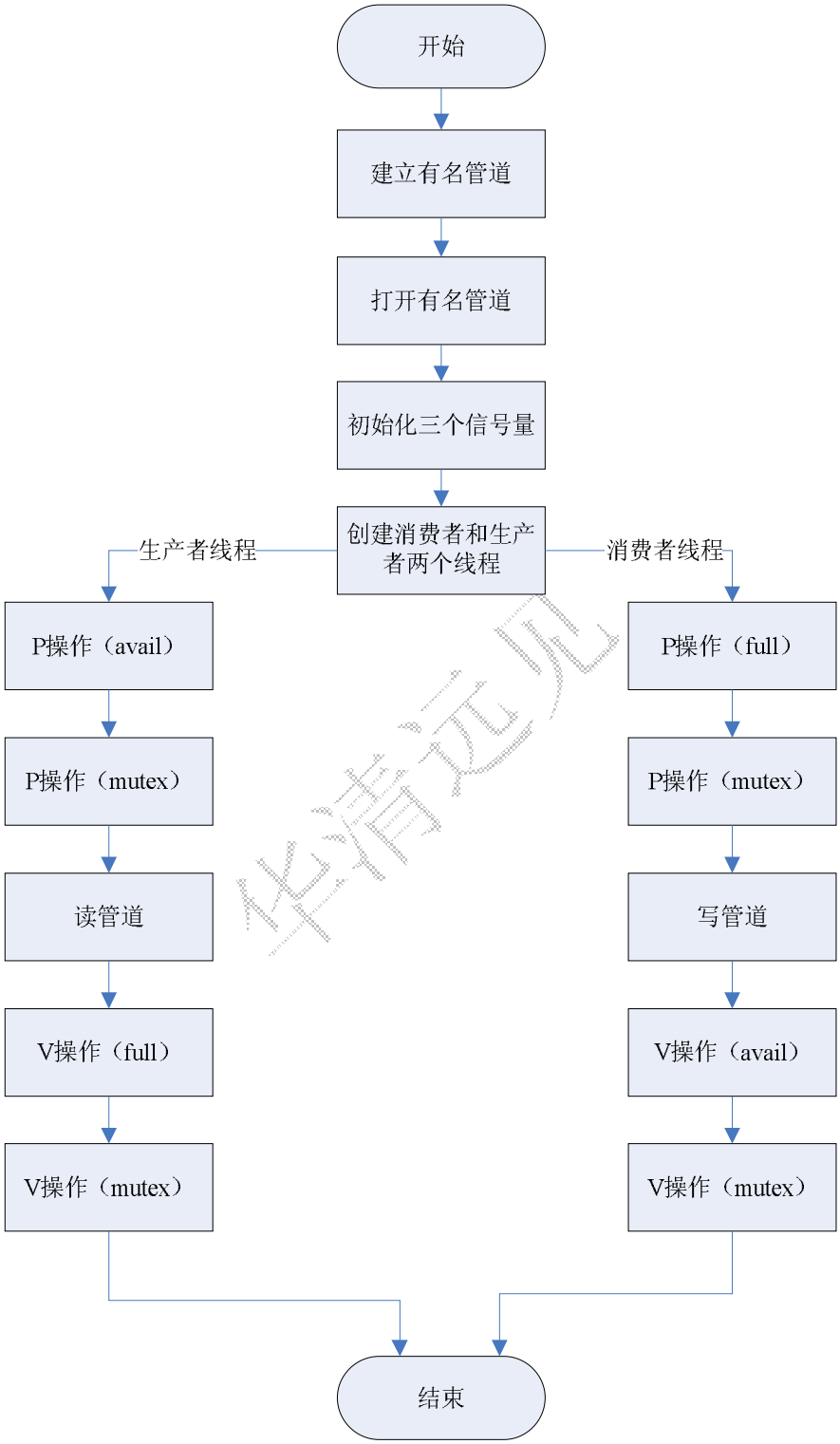

图 9.5 “生产者消费者”实验流程图

（3）编写代码

本实验代码如下：

```
/*product.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>
#include <sys/ipc.h>
#include <semaphore.h>
#include <fcntl.h>
#define FIFO "myfifo"
#define N 5
int lock_var;
time_t end_time;
char buf_r[100];
sem_t mutex,full,avail;
int fd;
void pthread1(void *arg);
void pthread2(void *arg);

int main(int argc, char *argv[])
{
     pthread_t id1,id2;
     pthread_t mon_th_id;
     int ret;
     end_time = time(NULL)+30;
/*创建有名管道*/
     if((mkfifo(FIFO,O_CREAT|O_EXCL)<0)&&(errno!=EEXIST))
          printf("cannot create fifoserver\n");
     printf("Preparing for reading bytes...\n");
     memset(buf_r,0,sizeof(buf_r));
/*打开管道*/
     fd=open(FIFO,O_RDWR|O_NONBLOCK,0);
     if(fd==-1)
     {
          perror("open");
          exit(1);
     }
```

```
/*初始化互斥信号量为 1*/
     ret=sem_init(&mutex,0,1);
/*初始化 avail 信号量为 N*/
     ret=sem_init(&avail,0,N);
/*初始化 full 信号量为 0*/
     ret=sem_init(&full,0,0);
     if(ret!=0)
     {
           perror("sem_init");
     }
/*创建两个线程*/
     ret=pthread_create(&id1,NULL,(void *)productor, NULL);
     if(ret!=0)
           perror("pthread cread1");
     ret=pthread_create(&id2,NULL,(void *)consumer, NULL);
     if(ret!=0)
           perror("pthread cread2");
     pthread_join(id1,NULL);
     pthread_join(id2,NULL);

     exit(0);
}

/*生产者线程*/
void productor(void *arg)
{
     int i,nwrite;
     while(time(NULL) < end_time){
/*P 操作信号量 avail 和 mutex*/
          sem_wait(&avail);
          sem_wait(&mutex);
/*生产者写入数据*/
          if((nwrite=write(fd,"hello",5))==-1)
          {
               if(errno==EAGAIN)
                     printf("The FIFO has not been read yet.Please try later\n");
          }
          else
               printf("write hello to the FIFO\n");
```

```
/*V操作信号量full和mutex*/
          sem_post(&full);
          sem_post(&mutex);
          sleep(1);
     }
}

/*消费者线程*/
void consumer(void *arg)
{
     int nolock=0;
     int ret,nread;
     while(time(NULL) < end_time){
/*P操作信号量full和mutex*/
         sem_wait(&full);
         sem_wait(&mutex);
         memset(buf_r,0,sizeof(buf_r));
         if((nread=read(fd,buf_r,100))==-1){
               if(errno==EAGAIN)
                     printf("no data yet\n");
         }
         printf("read %s from FIFO\n",buf_r);
/*V操作信号量avail和mutex*/
         sem_post(&avail);
         sem_post(&mutex);
         sleep(1);
     }
}
```

4．实验结果

运行该程序，得到如下结果：

```
[root@(none) tmp]#./exec
Preparing for reading bytes...
write hello to the FIFO
read hello from FIFO
write hello to the FIFO
read hello from FIFO
write hello to the FIFO
```

```
read hello from FIFO
write hello to the FIFO
read hello from FIFO
```

本 章 小 结

本章首先介绍了线程的基本概念、线程的分类和线程的发展历史，可以看出，线程技术已有了很大的进展。

接下来讲解了 Linux 中线程基本操作的 API 函数，包括线程的创建及退出，修改线程属性的操作，对每种操作都给出了简短的实例并加以说明。

再接下来，本章讲解了线程的控制操作，由于线程的操作必须包括线程间的同步互斥操作，包括互斥锁线程控制和信号量线程控制。

最后，本章的实验是一个经典的生产者——消费者问题，可以使用线程机制很好地实现，希望读者能够认真地编程实验，进一步理解多线程的同步、互斥操作。

思考与练习

通过查找资料，查看主流的嵌入式操作系统（如嵌入式 Linux，Vxworks 等）是如何处理多线程操作的？