嵌入式与移动开发系列

NITE 国家信息技术紧缺人才培养工程
National Information Technology Education Project
国家信息技术紧缺人才培养工程系列丛书

众多专家、厂商联合推荐 • 业界权威培训机构的经验总结

嵌入式Linux应用程序开发标准教程（第2版）

华清远见嵌入式培训中心 编著

提供36小时嵌入式专家讲座视频和教学课件

Embeded Linux Application Development

人民邮电出版社
POSTS & TELECOM PRESS

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 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

9.1 Linux 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复杂的上下文切换等动作。为了进一步减少处理机的空转时间，支持多处理器以及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。它是进程内独立的一条运行路线，处理器调度的最小单元，也可以称为轻量级进程。线程可以对进程的内存空间和资源进行访问，并与同一进程中的其他线程共享。因此，线程的上下文切换的开销比创建进程小很多。

同进程一样，线程也将相关的执行状态和存储变量放在线程控制表内。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。由此可知，多线程中的同步是非常重要的问题。在多线程系统中，进程与进程的关系如图 9.1 所示。

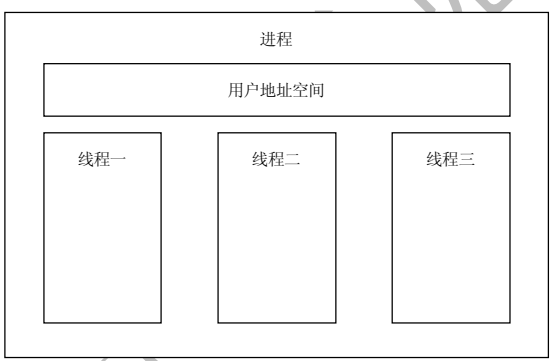


图 9.1　进程与线程关系

9.1.2 线程机制的分类和特性

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、调度和撤销等功能，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函数，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）轻量级进程。

轻量级进程是内核支持的用户线程，是内核线程的一种抽象对象。每个线程拥有一个或多个轻量级线程，而每个轻量级线程分别被绑定在一个内核线程上。

（3）内核线程。

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。

使用线程机制大大加快上下文切换速度而且节省很多资源。但是因为在用户态和内核态均要实现调度管理，所以会增加实现的复杂度和引起优先级翻转的可能性。一个多线程程序的同步设计与调试也会增加程序实现的难度。

9.2 Linux 线程编程

9.2.1 线程基本编程

这里要讲的线程相关操作都是用户空间中的线程的操作。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）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来等待线程的结束。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。

前面已提到线程调用 pthread_exit()函数主动终止自身线程。但是在很多线程应用中，经常会遇到在别的线程中要终止另一个线程的执行的问题。此时调用 pthread_cancel()函数实现这种功能，但在被取消的线程的内部需要调用 pthread_setcancel()函数和 pthread_setcanceltype()函数设置自己的取消状态，例如被取消的线程接收到另一个线程的取消请求之后，是接受还是忽略这个请求；如果接受，是立刻进行终止操作还是等待某个函数的调用等。

（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（其具体设置参见 9.2.3 小节），通常取为 NULL
	start_routine：线程函数的起始地址，是一个以指向 void 的指针作为参数和返回值的函数指针
	arg：传递给 start_routine 的参数
函数返回值	成功：0
	出错：返回错误码

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：线程结束时的返回值，可由其他函数如 pthread_join()来获取

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结束时的返回值（不为 NULL 时）
函数返回值	成功：0
	出错：返回错误码

表 9.4 列出了 pthread_cancel()函数的语法要点。

表 9.4　　pthread_cancel()函数语法要点

所需头文件	#include <pthread.h>
函数原型	int pthread_cancel((pthread_t th)
函数传入值	th：要取消的线程的标识符
函数返回值	成功：0
	出错：返回错误码

（3）函数使用。

以下实例中创建了 3 个线程，为了更好地描述线程之间的并行执行，让 3 个线程重用同一个执行函数。每个线程都有 5 次循环（可以看成 5 个小任务），每次循环之间会随机等待 1～10s 的时间，意义在于模拟每个任务的到达时间是随机的，并没有任何特定规律。

```
/* thread.c */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define THREAD_NUMBER      3              /*线程数*/
#define REPEAT_NUMBER      5              /*每个线程中的小任务数*/
#define DELAY_TIME_LEVELS  10.0           /*小任务之间的最大时间间隔*/

void *thrd_func(void *arg)
{ /* 线程函数例程 */
    int thrd_num = (int)arg;
    int delay_time = 0;
    int count = 0;

    printf("Thread %d is starting\n", thrd_num);
    for (count = 0; count < REPEAT_NUMBER; count++)
    {
        delay_time = (int)(rand() * DELAY_TIME_LEVELS/(RAND_MAX)) + 1;
        sleep(delay_time);
        printf("\tThread %d: job %d delay = %d\n",
                        thrd_num, count, delay_time);
    }
    printf("Thread %d finished\n", thrd_num);
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t thread[THREAD_NUMBER];
    int no = 0, res;
    void * thrd_ret;

    srand(time(NULL));

    for (no = 0; no < THREAD_NUMBER; no++)
    {
        /* 创建多线程 */
        res = pthread_create(&thread[no], NULL, thrd_func, (void*)no);
```

```
            if (res != 0)
            {
                printf("Create thread %d failed\n", no);
                exit(res);
            }
        }

        printf("Create treads success\n Waiting for threads to
finish...\n");
        for (no = 0; no < THREAD_NUMBER; no++)
        {
            /* 等待线程结束 */
            res = pthread_join(thread[no], &thrd_ret);
            if (!res)
            {
            printf("Thread %d joined\n", no);
            }
            else
            {
            printf("Thread %d join failed\n", no);
            }
        }
        return 0;
    }
```

以下是程序运行结果。可以看出每个线程的运行和结束是独立与并行的。

```
    $ ./thread
    Create treads success
    Waiting for threads to finish...
    Thread 0 is starting
    Thread 1 is starting
    Thread 2 is starting
           Thread 1: job 0 delay = 6
           Thread 2: job 0 delay = 6
           Thread 0: job 0 delay = 9
           Thread 1: job 1 delay = 6
           Thread 2: job 1 delay = 8
           Thread 0: job 1 delay = 8
           Thread 2: job 2 delay = 3
           Thread 0: job 2 delay = 3
```

```
        Thread 2: job 3 delay = 3
        Thread 2: job 4 delay = 1
Thread 2 finished
        Thread 1: job 2 delay = 10
        Thread 1: job 3 delay = 4
        Thread 1: job 4 delay = 1
Thread 1 finished
        Thread 0: job 3 delay = 9
        Thread 0: job 4 delay = 2
Thread 0 finished
Thread 0 joined
Thread 1 joined
Thread 2 joined
```

9.2.2 线程之间的同步与互斥

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同步与互斥问题。这里主要介绍 POSIX 中两种线程同步机制，分别为互斥锁和信号量。这两个同步机制可以互相通过调用对方来实现，但互斥锁更适合用于同时可用的资源是惟一的情况；信号量更适合用于同时可用的资源为多个的情况。

1．互斥锁线程控制

（1）函数说明。

互斥锁是用一种简单的加锁方法来控制对共享资源的原子操作。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被上锁的互斥锁，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，这把互斥锁保证让每个线程对共享资源按顺序进行原子操作。

互斥锁机制主要包括下面的基本函数。

- 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 3 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。默认属性为快速互斥锁。

（2）函数格式。

表 9.5 列出了 pthread_mutex_init()函数的语法要点。

表 9.5　　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	
	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：返回错误码	

表 9.6 列出了 pthread_mutex_lock()等函数的语法要点。

表 9.6　　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。

下面的实例是在 9.2.1 小节示例代码的基础上增加互斥锁功能，实现原本独立与无序的多个线程能够按顺序执行。

```
/*thread_mutex.c*/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define THREAD_NUMBER       3           /* 线程数 */
#define REPEAT_NUMBER       3           /* 每个线程的小任务数 */
#define DELAY_TIME_LEVELS 10.0         /*小任务之间的最大时间间隔*/
pthread_mutex_t mutex;
```

```
void *thrd_func(void *arg)
{
    int thrd_num = (int)arg;
    int delay_time = 0, count = 0;
    int res;
    /* 互斥锁上锁 */
    res = pthread_mutex_lock(&mutex);
    if (res)
    {
        printf("Thread %d lock failed\n", thrd_num);
        pthread_exit(NULL);
    }
    printf("Thread %d is starting\n", thrd_num);
    for (count = 0; count < REPEAT_NUMBER; count++)
    {
    delay_time = (int)(rand() * DELAY_TIME_LEVELS/(RAND_MAX)) + 1;
    sleep(delay_time);
    printf("\tThread %d: job %d delay = %d\n",
                            thrd_num, count, delay_time);
    }
    printf("Thread %d finished\n", thrd_num);
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t thread[THREAD_NUMBER];
    int no = 0, res;
    void * thrd_ret;

    srand(time(NULL));
    /* 互斥锁初始化 */
    pthread_mutex_init(&mutex, NULL);
    for (no = 0; no < THREAD_NUMBER; no++)
    {
        res = pthread_create(&thread[no], NULL, thrd_func, (void*)no);
        if (res != 0)
        {
            printf("Create thread %d failed\n", no);
            exit(res);
```

```
            }
        }
        printf("Create treads success\n Waiting for threads to
finish...\n");
        for (no = 0; no < THREAD_NUMBER; no++)
        {
            res = pthread_join(thread[no], &thrd_ret);
            if (!res)
            {
            printf("Thread %d joined\n", no);
            }
            else
            {
                printf("Thread %d join failed\n", no);
            }
            /* 互斥锁解锁 */
            pthread_mutex_unlock(&mutex);
        }
        pthread_mutex_destroy(&mutex);
        return 0;
    }
```

该实例的运行结果如下所示。这里 3 个线程之间的运行顺序跟创建线程的顺序相同。

```
    $ ./thread_mutex
    Create treads success
     Waiting for threads to finish...
    Thread 0 is starting
           Thread 0: job 0 delay = 7
           Thread 0: job 1 delay = 7
           Thread 0: job 2 delay = 6
    Thread 0 finished
    Thread 0 joined
    Thread 1 is starting
           Thread 1: job 0 delay = 3
           Thread 1: job 1 delay = 5
           Thread 1: job 2 delay = 10
    Thread 1 finished
    Thread 1 joined
```

```
Thread 2 is starting
        Thread 2: job 0 delay = 6
        Thread 2: job 1 delay = 10
        Thread 2: job 2 delay = 8
Thread 2 finished
Thread 2 joined
```

2．信号量线程控制

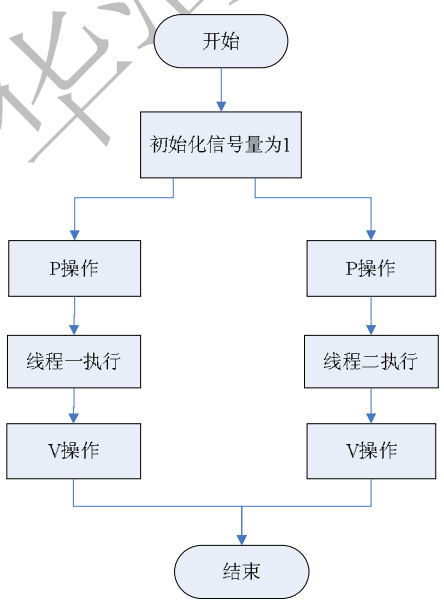
（1）信号量说明。

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 原子操作，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 原子操作的工作原理。

PV 原子操作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 原子操作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。



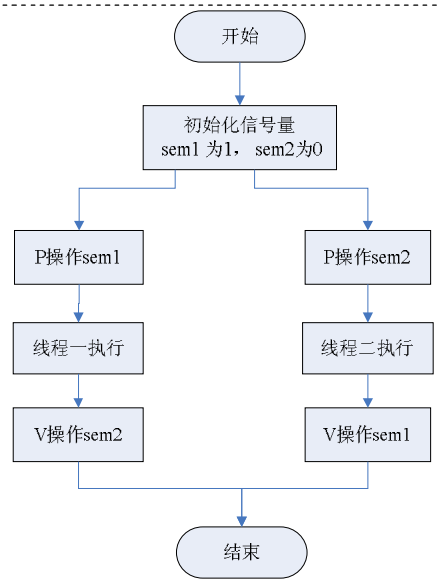


图 9.2　信号量互斥操作　　图 9.3　信号量同步操作

（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与同步。这里主要介绍几个常见函数。

- sem_init()用于创建一个信号量，并初始化它的值。
- sem_wait()和 sem_trywait()都相当于 P 操作，在信号量大于零时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来唤醒等待的进程。
- 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 9.7 列出了 sem_init()函数的语法要点。

表 9.7　　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：信号量指针
	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，就表示这个信号量是当前进程的局部信号量
	value：信号量初始化值
函数返回值	成功：0
	出错：−1

表 9.8 列出了 sem_wait()等函数的语法要点。

表 9.8　　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：信号量指针
函数返回值	成功：0
	出错：−1

（4）使用实例。

在前面已经通过互斥锁同步机制实现了多线程的顺序执行。下面的例子是用信号量同步机制实现 3 个线程之间的有序执行，只是执行顺序是跟创建线程的顺序相反。

```
/*thread_sem.c*/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define THREAD_NUMBER       3              /* 线程数 */
#define REPEAT_NUMBER       3              /* 每个线程中的小任务数 */
#define DELAY_TIME_LEVELS   10.0           /*小任务之间的最大时间间隔*/
sem_t sem[THREAD_NUMBER];

void *thrd_func(void *arg)
{
    int thrd_num = (int)arg;
    int delay_time = 0;
    int count = 0;
    /* 进行 P 操作 */
    sem_wait(&sem[thrd_num]);
    printf("Thread %d is starting\n", thrd_num);

    for (count = 0; count < REPEAT_NUMBER; count++)
    {
        delay_time = (int)(rand() * DELAY_TIME_LEVELS/(RAND_MAX)) + 1;
        sleep(delay_time);
        printf("\tThread %d: job %d delay = %d\n",
```

```
    thrd_num, count, delay_time);
        }

        printf("Thread %d finished\n", thrd_num);
        pthread_exit(NULL);
    }

    int main(void)
    {
        pthread_t thread[THREAD_NUMBER];
        int no = 0, res;
        void * thrd_ret;

        srand(time(NULL));
        for (no = 0; no < THREAD_NUMBER; no++)
        {
            sem_init(&sem[no], 0, 0);
            res = pthread_create(&thread[no], NULL, thrd_func, (void*)no);
            if (res != 0)
            {
                printf("Create thread %d failed\n", no);
                exit(res);
            }
        }

        printf("Create treads success\n Waiting for threads to
finish...\n");
        /* 对最后创建的线程的信号量进行 V 操作 */
        sem_post(&sem[THREAD_NUMBER - 1]);
        for (no = THREAD_NUMBER - 1; no >= 0; no--)
        {
           res = pthread_join(thread[no], &thrd_ret);
           if (!res)
           {
             printf("Thread %d joined\n", no);
            }
            else
            {
              printf("Thread %d join failed\n", no);
            }
```

```
            /* 进行 V 操作 */
            sem_post(&sem[(no + THREAD_NUMBER - 1) % THREAD_NUMBER]);
        }

        for (no = 0; no < THREAD_NUMBER; no++)
        {
            /* 删除信号量 */
            sem_destroy(&sem[no]);
        }

        return 0;
    }
```

该程序运行结果如下所示：

```
$ ./thread_sem
Create treads success
Waiting for threads to finish...
Thread 2 is starting
       Thread 2: job 0 delay = 9
       Thread 2: job 1 delay = 5
       Thread 2: job 2 delay = 10
Thread 2 finished
Thread 2 joined
Thread 1 is starting
       Thread 1: job 0 delay = 7
       Thread 1: job 1 delay = 4
       Thread 1: job 2 delay = 4
Thread 1 finished
Thread 1 joined
Thread 0 is starting
       Thread 0: job 0 delay = 10
       Thread 0: job 1 delay = 8
       Thread 0: job 2 delay = 9
Thread 0 finished
Thread 0 joined
```

9.2.3 线程属性

（1）函数说明。

pthread_create()函数的第二个参数（pthread_attr_t *attr）表示线程的属性。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、堆栈大小以及优先级。其中系统默认的属性为非绑定、非分离、缺省 1M 的堆栈以及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。而与之对应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特定的函数来完成的，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，最后调用 pthread_attr_destroy()函数对分配的属性结构指针进行清理和回收。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 9.9 列出了 pthread_attr_init()函数的语法要点。

表 9.9　　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：线程属性结构指针
函数返回值	成功：0
	出错：返回错误码

表 9.10 列出了 pthread_attr_setscope()函数的语法要点。

表 9.10　　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：线程属性结构指针

	scope	PTHREAD_SCOPE_SYSTEM：绑定
		PTHREAD_SCOPE_PROCESS：非绑定
函数返回值	成功：0	
	出错：−1	

表 9.11 列出了 pthread_attr_setdetachstate()函数的语法要点。

表 9.11　　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：返回错误码	

表 9.12 列出了 pthread_attr_getschedparam()函数的语法要点。

表 9.12　　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：线程属性结构指针
	param：线程优先级
函数返回值	成功：0
	出错：返回错误码

表 9.13 列出了 pthread_attr_setschedparam()函数的语法要点。

表 9.13　　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：线程属性结构指针
	param：线程优先级
函数返回值	成功：0
	出错：返回错误码

（3）使用实例。

下面的实例是在我们已经很熟悉的实例的基础上增加线程属性设置的功能。为了避免不必要的复杂性，这里就创建一个线程，这个线程具有绑定和分离属性，而且主线程通过一个 finish_flag 标志变量来获得线程结束的消息，而并不调用 pthread_join()函数。

```
/*thread_attr.c*/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define REPEAT_NUMBER        3          /* 线程中的小任务数 */
#define DELAY_TIME_LEVELS    10.0       /* 小任务之间的最大时间间隔 */
int finish_flag = 0;

void *thrd_func(void *arg)
{
    int delay_time = 0;
    int count = 0;

    printf("Thread is starting\n");
    for (count = 0; count < REPEAT_NUMBER; count++)
    {
        delay_time = (int)(rand() * DELAY_TIME_LEVELS/(RAND_MAX)) + 1;
        sleep(delay_time);
        printf("\tThread : job %d delay = %d\n", count, delay_time);
    }

    printf("Thread finished\n");
    finish_flag = 1;
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t thread;
    pthread_attr_t attr;
    int no = 0, res;
    void * thrd_ret;

    srand(time(NULL));
    /* 初始化线程属性对象 */
    res = pthread_attr_init(&attr);
    if (res != 0)
    {
        printf("Create attribute failed\n");
```

```
            exit(res);
        }
        /* 设置线程绑定属性 */
        res = pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
        /* 设置线程分离属性 */
        res            +=            pthread_attr_setdetachstate(&attr,
PTHREAD_CREATE_DETACHED);
        if (res != 0)
        {
            printf("Setting attribute failed\n");
            exit(res);
        }

        res = pthread_create(&thread, &attr, thrd_func, NULL);
        if (res != 0)
        {
            printf("Create thread failed\n");
            exit(res);
        }
        /* 释放线程属性对象 */
        pthread_attr_destroy(&attr);
        printf("Create tread success\n");

        while(!finish_flag)
        {
            printf("Waiting for thread to finish...\n");
            sleep(2);
        }
        return 0;
    }
```

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

```
    $ ./thread_attr
    Create tread success
    Waiting for thread to finish...
    Thread is starting
    Waiting for thread to finish...
           Thread : job 0 delay = 3
    Waiting for thread to finish...
```

```
        Thread : job 1 delay = 2
Waiting for thread to finish...
Waiting for thread to finish...
Waiting for thread to finish...
Waiting for thread to finish...
        Thread : job 2 delay = 9
Thread finished

/* 程序运行之前 */
$ free
             total       used       free     shared    buffers     cached
Mem:        255556     191940      63616         10       5864      61360
-/+ buffers/cache:     124716     130840
Swap:       377488      18352     359136

/* 程序运行之中 */
$ free
             total       used       free     shared    buffers     cached
Mem:        255556     191948      63608         10       5888      61336
-/+ buffers/cache:     124724     130832
Swap:       377488      18352     359136

/* 程序运行之后 */
$ free
             total       used       free     shared    buffers     cached
Mem:        255556     191940      63616         10       5904      61320
-/+ buffers/cache:     124716     130840
Swap:       377488      18352     359136
```

可以看到，线程在运行结束后就收回了系统资源，并释放内存。

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。通过学习经典的“生产者消费者”问题的实验，读者可以进一步熟悉 Linux 中的多线程编程，并且掌握用信号量处理线程间的同步和互斥问题。

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。他们分别不停地把产品放入缓冲区和从缓冲区中拿走产品。一个生产者在缓冲区满的时候必须等待，一个消费者在缓冲区空的时候也必须等待。另外，因为缓冲区是临界资源，所以生产者和消费者之间必须互斥执行。它们之间的关系如图 9.4 所示。

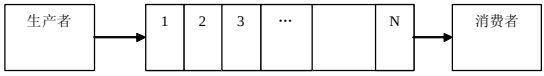


图 9.4　生产者消费者问题描述

这里要求使用有名管道来模拟有限缓冲区，并且使用信号量来解决“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 表示有界缓冲区中的空单元数，初始值为 N；full 表示有界缓冲区中非空单元数，初始值为 0；mutex 是互斥信号量，初始值为 1。

（2）画出流程图。

本实验流程图如图 9.5 所示。

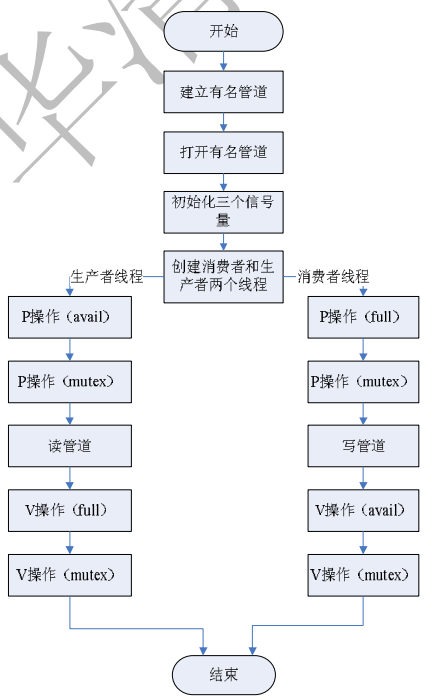


图 9.5　“生产者—消费者”实验流程图

（3）编写代码

本实验的代码中采用的有界缓冲区拥有 3 个单元，每个单元为 5 个字节。为了尽量体现每个信号量的意义，在程序中生产过程和消费过程是随机（采取 0～5s 的随机时间间隔）进行的，而且生产者的速度比消费者的速度平均快两倍左右（这种关系可以相反）。生产者一次生产一个单元的产品（放入“hello”字符串），消费者一次消费一个单元的产品。

```
    /*producer-customer.c*/
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <fcntl.h>
    #include <pthread.h>
    #include <errno.h>
    #include <semaphore.h>
    #include <sys/ipc.h>

    #define MYFIFO          "myfifo"    /* 缓冲区有名管道的名字 */
    #define BUFFER_SIZE      3          /* 缓冲区的单元数 */
    #define UNIT_SIZE        5          /* 每个单元的大小 */
    #define RUN_TIME         30         /* 运行时间 */
    #define DELAY_TIME_LEVELS    5.0    /* 周期的最大值 */

    int fd;
    time_t end_time;
    sem_t mutex, full, avail;           /* 3 个信号量 */

    /*生产者线程*/
    void *producer(void *arg)
    {
        int real_write;
        int delay_time = 0;

        while(time(NULL) < end_time)
        {
            delay_time = (int)(rand() * DELAY_TIME_LEVELS/(RAND_MAX) / 2.0)
+ 1;
            sleep(delay_time);
            /*P 操作信号量 avail 和 mutex*/
            sem_wait(&avail);
```

```
            sem_wait(&mutex);
            printf("\nProducer: delay = %d\n", delay_time);
            /*生产者写入数据*/
            if ((real_write = write(fd, "hello", UNIT_SIZE)) == -1)
            {
                if(errno == EAGAIN)
                {
                    printf("The FIFO has not been read yet.Please try
later\n");
                }
            }
            else
            {
                printf("Write %d to the FIFO\n", real_write);
            }

            /*V 操作信号量 full 和 mutex*/
            sem_post(&full);
            sem_post(&mutex);
        }
        pthread_exit(NULL);
    }
    /* 消费者线程*/
    void *customer(void *arg)
    {
        unsigned char read_buffer[UNIT_SIZE];
        int real_read;
        int delay_time;

        while(time(NULL) < end_time)
        {
            delay_time = (int)(rand() * DELAY_TIME_LEVELS/(RAND_MAX)) + 1;
            sleep(delay_time);

            /*P 操作信号量 full 和 mutex*/
            sem_wait(&full);
            sem_wait(&mutex);
            memset(read_buffer, 0, UNIT_SIZE);
                    printf("\nCustomer: delay = %d\n", delay_time);
```

```
        if ((real_read = read(fd, read_buffer, UNIT_SIZE)) == -1)
        {
            if (errno == EAGAIN)
            {
                printf("No data yet\n");
            }
        }
        printf("Read %s from FIFO\n", read_buffer);
        /*V 操作信号量 avail 和 mutex*/
        sem_post(&avail);
        sem_post(&mutex);
    }
    pthread_exit(NULL);
}

int main()
{
    pthread_t thrd_prd_id,thrd_cst_id;
    pthread_t mon_th_id;
    int ret;

    srand(time(NULL));
    end_time = time(NULL) + RUN_TIME;
    /*创建有名管道*/
    if((mkfifo(MYFIFO, O_CREAT|O_EXCL) < 0) && (errno != EEXIST))
    {
        printf("Cannot create fifo\n");
        return errno;
    }
    /*打开管道*/
    fd = open(MYFIFO, O_RDWR);
    if (fd == -1)
    {
        printf("Open fifo error\n");
        return fd;
    }
    /*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, BUFFER_SIZE);
```

```
    /*初始化 full 信号量为 0*/
    ret += sem_init(&full, 0, 0);
    if (ret != 0)
    {
        printf("Any semaphore initialization failed\n");
        return ret;
    }
    /*创建两个线程*/
    ret = pthread_create(&thrd_prd_id, NULL, producer, NULL);
    if (ret != 0)
    {
        printf("Create producer thread error\n");
        return ret;
    }
    ret = pthread_create(&thrd_cst_id, NULL, customer, NULL);
    if(ret != 0)
    {
        printf("Create customer thread error\n");
        return ret;
    }
    pthread_join(thrd_prd_id, NULL);
    pthread_join(thrd_cst_id, NULL);
    close(fd);
    unlink(MYFIFO);
    return 0;
}
```

4．实验结果

运行该程序，得到如下结果：

```
$ ./producer_customer
……
Producer: delay = 3
Write 5 to the FIFO

Customer: delay = 3
Read hello from FIFO

Producer: delay = 1
Write 5 to the FIFO
```

```
Producer: delay = 2
Write 5 to the FIFO

Customer: delay = 4
Read hello from FIFO

Customer: delay = 1
Read hello from FIFO

Producer: delay = 2
Write 5 to the FIFO
……
```

9.4 本章小结

本章首先介绍了线程的基本概念、线程的分类和特性以及线程的发展历程。

接下来讲解了 Linux 中线程库的基本操作函数，包括线程的创建、退出和取消等，通过实例程序给出了比较典型的线程编程框架。

再接下来，本章讲解了线程的控制操作。在线程的操作中必须实现线程间的同步和互斥，其中包括互斥锁线程控制和信号量线程控制。后面还简单描述了线程属性相关概念、相关函数以及比较简单的典型实例。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和互斥操作。

9.5 思考与练习

1．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。

2．通过线程实现串口通信。

3．通过线程和网络编程实现网上聊天程序。