## 第11课 讲稿（多线程编程）

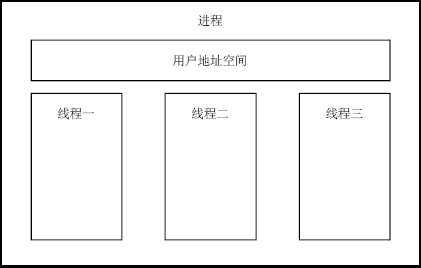
1. **Linux 线程概述**
2. 线程概述

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（Lightweight process）。**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。

* 上下文切换：CPU的控制权由运行任务转移到另外一个就绪任务时所发生的事件，主要包括保存当前任务的运行环境、恢复将要运行任务的运行环境。

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，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，进程与线程的关系如下图所示：



一个标准的线程由**线程ID，当前指令**[**指针**](http://baike.baidu.com/view/159417.htm)**(PC），**[**寄存器**](http://baike.baidu.com/view/6159.htm)**集合和**[**堆栈**](http://baike.baidu.com/view/93201.htm)组成。

一个线程可以创建和撤消另一个线程，同一进程中的多个线程之间可以并发执行。

由于线程之间的相互制约，致使线程在运行中呈现出间断性。线程也有[就绪](http://baike.baidu.com/view/654230.htm)、[阻塞](http://baike.baidu.com/view/497285.htm)和[运行](http://baike.baidu.com/view/1026025.htm)三种基本状态。每一个程序都至少有一个线程，若程序只有一个线程，那就是程序本身。在单个程序中同时运行多个线程完成不同的工作，称为[多线程](http://baike.baidu.com/view/65706.htm)。如聊天过程中，server和client可以在各自的进程中创建2个线程，一个负责接受信息，一个负责发送信息，即可实现同时收发。

1. **Linux 线程编程**
2. 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，这里使用的函数是pthread\_create()。在线程创建以后，**新创建的线程开始运行相关的线程函数**（线程函数由第3个和第4个参数确定），**原来的线程继续运行下一行代码**，**新创建的线程在运行完线程函数后，该线程也就退出了**，这也是线程退出的一种方法。

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。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，经常会遇到在一个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\_setcancelstate()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。

1. 函数格式。

表1列出了pthread\_create()函数的语法要点。

|  |  |
| --- | --- |
| 表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：指向一个内存地址，用来记录新创建线程的线程标识符 |
| attr：线程属性设置，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 |
| 返回值 | 成功：0 |
| 出错：返回错误码 |

表2列出了pthread\_exit()函数的语法要点。

|  |  |
| --- | --- |
| 表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，一般为0或NULL |

表3列出了pthread\_join()函数的语法要点。

|  |  |
| --- | --- |
| 表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标识符，如果不需要保存，可设置为NULL |
| 返回值 | 成功：0 |
| 出错：返回错误码 |

表4列出了pthread\_cancel()函数的语法要点。

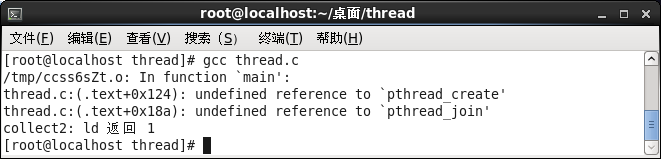
|  |  |
| --- | --- |
| 表4 pthread\_cancel()函数语法要点 | |
| 所需头文件 | #include <pthread.h> |
| 函数原型 | int pthread\_cancel((pthread\_t th) |
| 函数传入值 | th：要取消的线程的标识符 |
| 返回值 | 成功：0 |
| 出错：返回错误码 |

1. 函数使用。

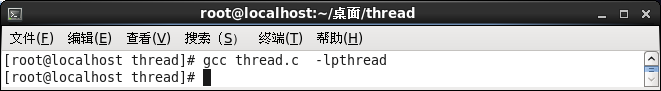
以下实例中创建了3个线程，为了更好地描述线程之间的并行执行，让3个线程重用同一个执行函数。每个线程都有5次循环（可以看成5个小任务），每次循环之间会随机等待1-10s的时间，意义在于模拟每个任务的到达时间是随机的，并没有任何特定规律。

|  |
| --- |
| /\* 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 /\*小任务之间的最大时间间隔\*/  /\* 线程函数\*/  void \*thrd\_func(void \*arg)  {  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++)  {  /\* 得到1-10的随机数 \*/  delay\_time = rand()% DELAY\_TIME\_LEVELS + 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]; /\* 记录线程标识符 \*/  int no = 0, res;  srand(time(0));  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], NULL);  if (!res)  {  printf("Thread %d joined\n", no);  }  else  {  printf("Thread %d join failed\n", no);  }  }  return 0;  } |

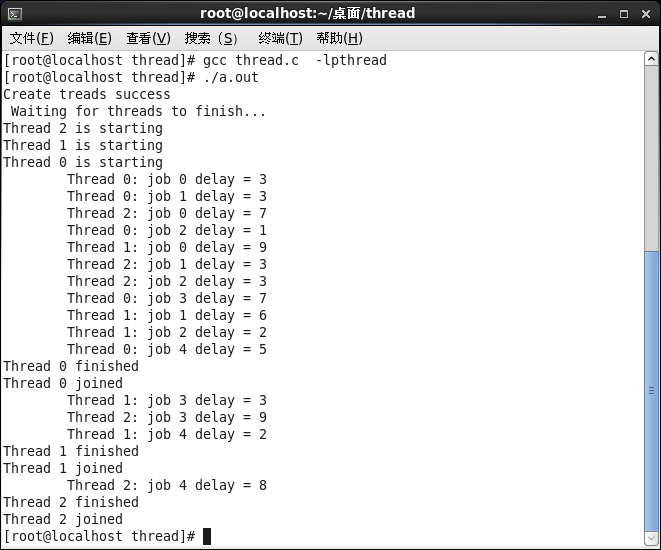
【注意】编译时如果出现如下提示：



【原因】因为pthread 库不是 Linux 系统默认的库，连接时需要使用静态库 libpthread.a，所以在使用pthread\_create()创建线程，以及调用 pthread\_atfork()函数建立fork处理程序时，需要在编译中添加-lpthread参数。



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。



【注意】线程的创建顺序是一定的，但是执行顺序不一定。根据for语句，线程创建的顺序为thread[0]→ thread[1]→ thread[2]，但是执行的时候可能是thread[0]→ thread[1]→ thread[2]也可能是thread[2]→ thread[1]→ thread[0]，或者其他顺序，具体顺序由系统调度决定。另外，由于thread\_join()是阻塞等待函数，所以不论线程结束的顺序如何，join的顺序一定是thread[0]→ thread[1]→ thread[2]。

1. 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。这里主要介绍互斥锁和信号量两种同步机制，这两种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。这里只介绍信号量控制。

1. 信号量线程控制
2. 信号量说明。

前面已经讲到，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。

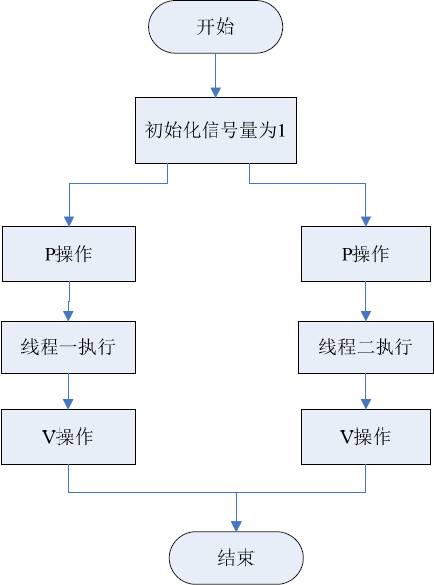
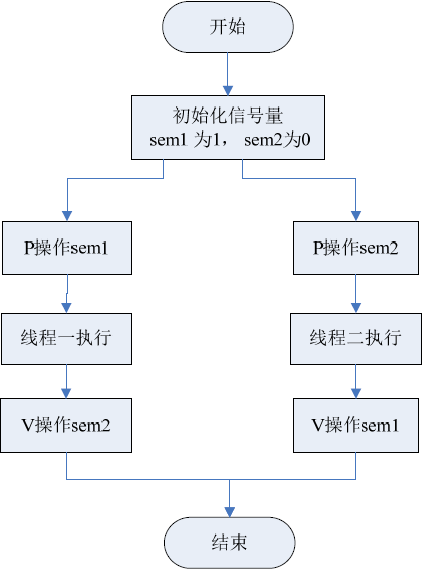
PV原子操作是对整数计数器信号量sem的操作。一次P操作使sem减1，而一次V操作使sem加1。

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：

* 互斥：指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。
* 同步：指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。

若用于互斥，几个进程（或线程）往往只设置一个信号量sem，它们的操作流程如下左图所示，此时依然无法控制执行的顺序：

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，它们的操作流程如上右图所示。

1. 函数说明。

Linux实现了POSIX的无名信号量，主要用于线程或具有亲缘关系的进程间的互斥与同步。

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0时它们都能将信号量的值减1。两者的区别在于若信号量小于0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1，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。

1. 函数格式。

表7列出了sem\_init()函数的语法要点。

|  |  |
| --- | --- |
| 表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。这个值取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 |
| 返回值 | 成功：0 |
| 出错：-1 |

表8列出了sem\_wait()等其他函数的语法要点。

|  |  |
| --- | --- |
| 表8 sem\_wait()等其他函数语法要点 | |
| 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 |
| 返回值 | 成功：0 |
| 出错：-1 |

1. 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 5 /\* 每个线程中的小任务数 \*/  #define DELAY\_TIME\_LEVELS 10 /\*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;  /\* 对信号量进行P操作 \*/  sem\_wait(&sem[thrd\_num]);  printf("Thread %d is starting\n", thrd\_num);  for (count = 0; count < REPEAT\_NUMBER; count++)  {  /\* 得到1-10的随机数 \*/  delay\_time = rand()% DELAY\_TIME\_LEVELS + 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3个初值为0的局部信号量\*/  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);  }  /\* 对前一个线程进行V操作 \*/  sem\_post(&sem[(no - 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：

