**浙大城市学院实验报告**

课程名称 操作系统原理实验

实验项目名称 实验四-线程的创建和管理

学生姓名 司晨旭 专业班级 计算2002 学号 32001019

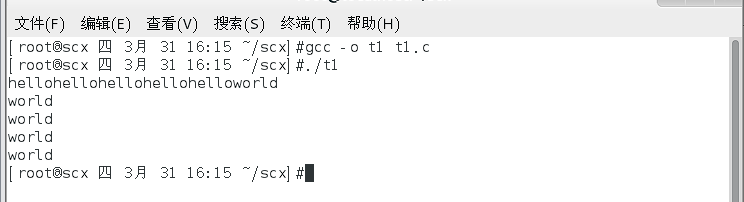
**实验步骤:**

1. **基本概念**
2. Linux并不存在真正的线程，因为Linux下没有给线程设计专有的结构体，Linux的线程是使用进程模拟的，线程可看成是轻量级的进程。创建一个线程的资源成本小，工作效率高。
3. Linux系统下的多线程遵循POSIX线程接口，称为pthread。POSIX标准族中包含有线程标准，有关的系统调用名字以“pthread\_”打头。编写Linux下的多线程程序，**需要使用头文件pthread.h，链接这些线程函数库时要使用编译器命令的“-lpthread”选项**。
4. **使用线程的例程**
5. 调试并运行给出的"hello world"单线程程序，观察输出结果，并分析原因。

|  |  |
| --- | --- |
| //"hello world"单线程程序  # include <stdio.h>  const NUM = 5;  int main()  {  p\_msg("hello");  p\_msg("world\n");  } | void p\_msg(char \*m)  {  int i;  for(i = 0 ; i < NUM ; i ++)  {  printf("%s",m);  fflush(stdout);  sleep(1);  }  } |

编译链接通过后，多次运行例程，观察进程并发执行结果，并思考下述问题：

1. 此例程使用函数调用，理解函数调用是顺序执行的；



如图所示，在主函数中优先执行p\_msg(“hello”)输出五个hello，然后执行p\_msg(“world\n”),输出5个world\n

1. 理解fflush( )的作用。

fflush的作用为清空当前进程的缓冲区，当删除代码中的fflush,程序运行时，hello并不是每个一秒输出一个，而是一下子五个hello全部输出，这是因为hello一直在缓冲区中，遇见\n后一下子全部输出出来

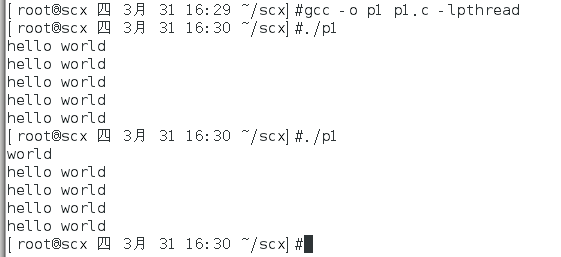
1. 如果希望一个程序能同时并发执行很多函数，它可以通过创建多个线程实现，只要将上面的程序做两个小的改动，即可让程序并发执行两个函数。下面的例程是一个简单“hello world”多线程程序。

|  |  |
| --- | --- |
| //"hello world"多线程程序  #include <stdio.h>  #include <pthread.h>  main()  {  pthread\_t t1 , t2;  void \*p\_msg(char \*);    **pthread\_create**(&t1,NULL,p\_msg,(void \* )"hello");  **pthread\_create**(&t2,NULL,p\_msg,(void \* )"world\n");  pthread\_join(t1, NULL);  pthread\_join(t2, NULL);  } | void \*p\_msg(char \*m)  {  char \*cp =(char \*) m;  int i;  for(i = 0 ; i < 5 ; i ++)  {  printf("%s",m);  fflush(stdout);  sleep(1);  }  return NULL;  } |

编译链接通过后，多次运行例程，观察进程和线程并发执行结果，并思考下述问题：

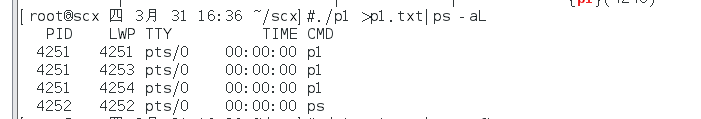
1. 从输出内容，观察线程的并发执行。

我的运行结果如下



1. 运行例程，用ps -aL、ps -efL和pstree -p观察线程信息。

ps -aL运行结果如下

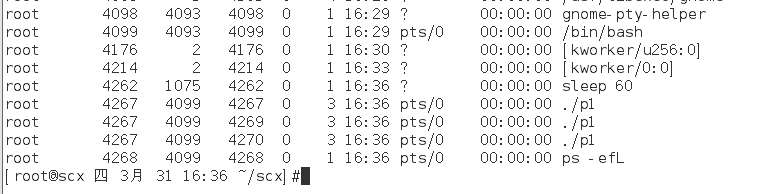




黄色部分为线程信息

ps-efL运行结果如下

运行结果很多，只给出线程相关部分





pstree -p 运行结果如下



（4242）表示主进程

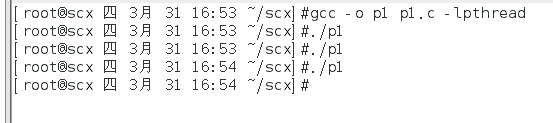
{}括起来的表示两个线程

1. 理解pthread\_create( )函数的作用。

pthread\_create是类Unix操作系统创建线程的函数。它的作用是创建线程（实际上就是确定调用该线程函数的入口点），在线程创建以后，就开始运行相关的线程函数

1. 理解pthread\_join( )函数的作用。尝试将pthread\_join( )删除，观察运行结果的变化。

运行结果如下



可以看到并没有任何的输出，原因为：

主进程的存在时间过短（或者说两个线程因为sleep函数存在的时间太长了），导致主进程提前结束掉了，线程在主进程结束时也随之消亡掉了。

故pthread()\_join的作用为：

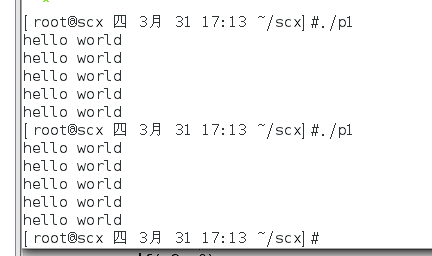
使得当前进程阻塞，直到指定的线程thread结束时，进程才能继续运行。如果指定的线程已经结束，函数立即返回。

1. 改写代码，使同样的输出内容由并发的父子进程实现，体会进程和线程概念的差异。

改写后的代码如下

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include<sys/types.h>  #include<sys/wait.h>  main()  {  int p1,p2;  while((p1=fork())==-1);  if(p1==0)  {  p\_msg((void \*)"hello ");    }  else  {  while((p2=fork())==-1);  if(p2==0)  {  p\_msg((void \*)"world\n");  }  else  {  wait(0);  }  }    }  void p\_msg(void \*m)  {  int i;  for(i = 0 ; i < 5 ; i ++)  {  printf("%s",m);  fflush(stdout);//清除文件缓冲区，文件以写方式打开时将缓冲区内容写入文件  sleep(1);  }  return NULL;  } |

运行结果如下



对进程和线程的差异的理解：一个进程可能由多个线程组成，这些线程共享他们的进程的pid和缓冲区

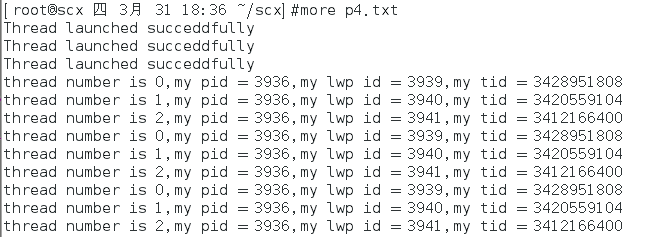
1. 运行程序，观察线程并发执行的效果，及关于线程的各种ID。

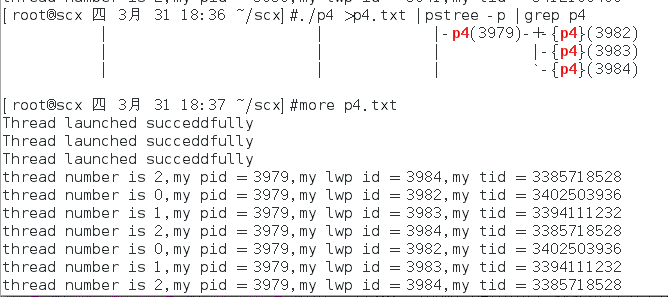
|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <sys/syscall.h>  void \*PrintThread(void \* );#define Num\_Threads 3  int main(){ int i,ret; pthread\_t a\_thread; int ThdNum[Num\_Threads];  for (i = 0; i < Num\_Threads; i++) ThdNum[i] = i;  for(i = 0; i < Num\_Threads;i++)  {  ret = pthread\_create(&a\_thread,NULL,PrintThread,(void \*)&ThdNum[i]);  if (ret == 0)  printf("Thread launched succeddfully\n");  }  i = getchar();  return (0);  }  void \*PrintThread(void \* num)  {  int i;  for(i = 0;i < 3; i++)  {  printf("thread number is %d,my pid = %d,my lwp id = %d,my tid = %u\n",\*((int \*)num),**getpid(),syscall(SYS\_gettid),pthread\_self()**);  sleep(1);  }  return NULL;//exit(0);  } |

编译链接通过后，多次运行程序，观察多线程并发运行，并思考：

1. 分析输出内容的顺序不确定的原因；

运行结果如下



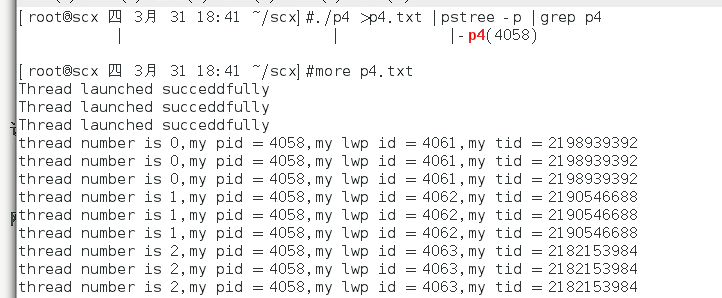


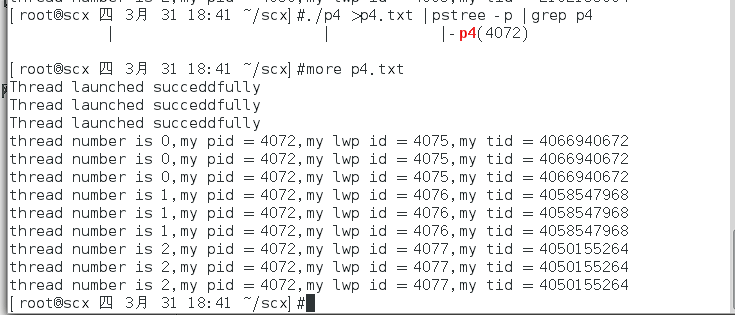
可以看到两次的运行结果并不相同

原因是三个线程也是并发执行的。因此执行顺序并不确定。

1. 删除sleep(1);观察输出内容的变化，并分析原因；

删除sleep（1）后，运行结果如下







两次执行结果都相同，且可以看到线程存在的时间很短，很难被捕捉到

原因：三个线程仍然是并发进行的，但由于线程存在的时间很短，执行的时间很快，因此占不到一个时间片，结果总是顺序执行的。

注意可能会出现120，201的运行结果。但可以发现仍然是顺序执行的。

1. 分析输出内容，理解线程各种id值的含义。

线程各id含义如下

pid表示线程的父进程的进程id

lwp id表示轻量级线程号

tid表示内部的线程号

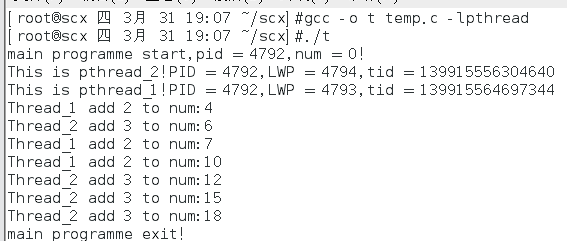
1. 运行下述例程，理解POSIX互斥锁的简单使用。

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  #include <pthread.h>  #include <errno.h>  #include <sys/types.h>  #include <sys/syscall.h>  /\*全局变量\*/  int gnum = 0;  /\*互斥量 \*/  **pthread\_mutex\_t mutex**;  /\*声明线程运行服务程序\*/  static void pthread\_add2 (void);  static void pthread\_add3 (void);  pid\_t gettid() {  return syscall(SYS\_gettid);  }  int main (void)  {  /\*线程的标识符\*/  pthread\_t pt\_1 = 0;  pthread\_t pt\_2 = 0;  int ret = 0;  printf ("main programme start,pid = %d,num = %d!\n",getpid(),gnum);  /\*互斥初始化\*/  **pthread\_mutex\_init (&mutex, NULL);**  /\*分别创建线程1、2\*/  ret = pthread\_create( &pt\_1, NULL,(void \*)pthread\_add2, NULL);  ret = pthread\_create( &pt\_2, NULL,(void \*)pthread\_add3, NULL);  /\*等待线程1、2的结束\*/  pthread\_join (pt\_1, NULL);  pthread\_join (pt\_2, NULL);  printf ("main programme exit!\n");  return 0;  }  /\*线程1的服务程序\*/  static void pthread\_add2 (void)  {  int i = 0;  printf ("This is pthread\_1!PID = %d,LWP = %d,tid = %lu\n",getpid(),gettid(),pthread\_self());  for( i=0; i<3; i++ ){  //**pthread\_mutex\_lock(&mutex);** /\*获取互斥锁\*/  **gnum++; sleep (1); gnum++;**/\*临界资源\*/  printf ("Thread\_1 add 2 to num:%d\n",gnum);  //**pthread\_mutex\_unlock(&mutex);** /\*释放互斥锁\*/  }  pthread\_exit ( NULL );  }  /\*线程2的服务程序\*/  static void pthread\_add3 (void)  {  int i = 0;  printf ("This is pthread\_2!PID = %d,LWP = %d,tid = %lu\n",getpid(),gettid(),pthread\_self());  for( i=0; i<4; i++ ) {  //**pthread\_mutex\_lock(&mutex);** /\*获取互斥锁\*/  **gnum++; sleep (1);gnum++; sleep (1);gnum++;** /\*临界资源\*/  printf ("Thread\_2 add 3 to num:%d\n",gnum);  //**pthread\_mutex\_unlock(&mutex);** /\*释放互斥锁\*/  }  pthread\_exit ( 0 );  } |

编译链接通过后，多次运行程序，观察多线程并发运行，并思考：

1. 保留注释，运行程序，可观察到输出混乱。请分析出现混乱的原因。

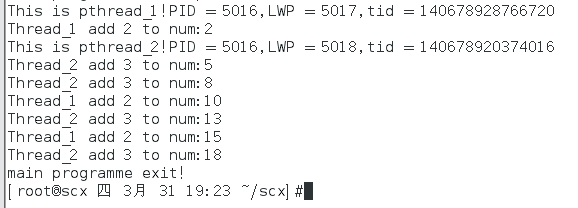
运行结果如下



两个线程是并发进行的，但是临界资源gnum并没有得到保护，也就是说，两个线程在同时对gnum进行操作，当第一个线程执行完gnum++之后，又转到了第二个线程执行gunm++，因此造成了输出结果的混乱

1. 删除sleep函数，运行程序，可观察到混乱现象消失了。思考这样是否从根本上彻底解决问题？

删除sleep（）函数后，运行结果如下

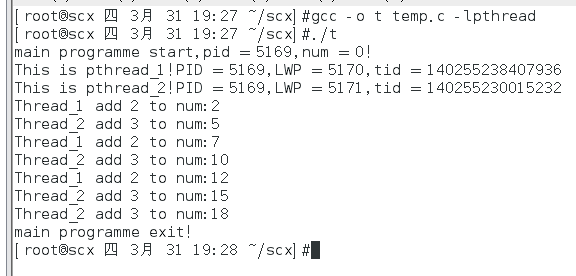


可以发现混乱现象似乎消失了，原因是删除了sleep（）函数之后，在使用公共资源gnum时，几乎没有了线程的切换。

但是并没有从根本上解决问题，假如时间片正好在使用公共资源时结束，线程不得不切换，因此还是会造成混乱现象。

1. 恢复sleep函数，删除注释，启用互斥锁，观察运行结果的变化，理解互斥锁的作用；

恢复sleep，启用互斥锁之后的运行结果如下。



互斥锁的作用：保护了临界资源，使得在一个线程使用完临界资源之前，其他的线程无法使用这个线程。

1. 掌握互斥锁的基本操作：初始化，加锁（P操作）和解锁（V操作）；

在本列程中，初始化代码为：pthread\_mutex\_init (&mutex, NULL);

加锁（P操作代码为）：pthread\_mutex\_lock(&mutex);P操作为将信号量减1，当这个值小于0，则阻塞当前指令

解锁（V操作代码为）: pthread\_mutex\_unlock(&mutex);V操作为信号量+1，当这个值小于等于0，则释放阻塞的进程或者线程。

1. 各线程输出内容不再混乱，和删除sleep函数效果类似，但是原因不同。请分析各自的原因。

互斥锁输出内容不再混乱原因：

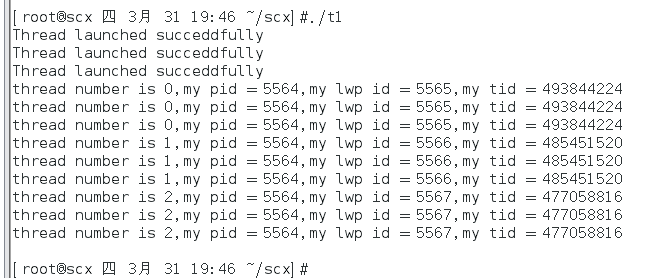
保护了临界资源，使得在一个线程使用完临界资源之前，其他的线程无法使用这个线程。

删除sleep（）后输出内容不再混乱原因：

在使用公共资源gnum时，由于当前线程的执行速度较快，几乎没有了线程的切换。

1. 修改例程3，使用互斥锁，使各线程的输出内容不交错。

首先先附上运行结果



修改后的例程3代码

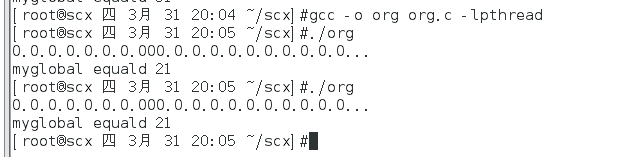
|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  //#include <sys/types.h>  #include <sys/syscall.h>  #include <pthread.h>  //#include <unistd.h>  void \*PrintThread(void \* );  pthread\_mutex\_t mutex;  #define Num\_Threads 3  int main()  {  int i,ret;  pthread\_t a\_thread;  int ThdNum[Num\_Threads];  pthread\_mutex\_init (&mutex, NULL);  for (i = 0; i < Num\_Threads; i++)  ThdNum[i] = i;  for(i = 0; i < Num\_Threads;i++)  {  ret = pthread\_create(&a\_thread,NULL,PrintThread,(void \*)&ThdNum[i]);  if (ret == 0)  printf("Thread launched succeddfully\n");  }  i = getchar();  return (0);  }  void \*PrintThread(void \* num)  {  int i;  pthread\_mutex\_lock(&mutex);  for(i = 0;i < 3; i++)  {  printf("thread number is %d,my pid = %d,my lwp id = %d,my tid = %u\n",\*((int \*)num),getpid(),syscall(SYS\_gettid),pthread\_self());  sleep(1);  }  pthread\_mutex\_unlock(&mutex);  return NULL;  } |

1. 练习题：运行教材习题4.9中的程序，解释原程序未获得期望值的原因。模仿例程4，使用POSIX互斥锁，保证运行结果获得期望值。

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <errno.h>  #include <unistd.h>  int myglobal=10;  void \*thread\_function(void \*arg)  {  int i,j;  for(i = 0; i <20; i++){  j = myglobal;  j++;  printf(".");  fflush(stdout);  usleep(1000);  myglobal = j;  }  return NULL;  }  int main(){  pthread\_t mythread;  int i;  if( pthread\_create(&mythread,NULL,thread\_function,NULL))  {  printf("error creating thread!\n");  abort();  }  for(i = 0; i <20; i++)  {  myglobal++;  printf("O");  fflush(stdout);  usleep(1000);  }  if( pthread\_join(mythread,NULL))  {  printf("error joining thread!\n");  abort();  }  printf("\nmyglobal equald %d\n",myglobal);  exit(0);  } |

观察程序加锁前后的变化，理解互斥锁的使用

原程序运行结果如下



未获得想要的原因时，主进程和生成的线程并发进行，两个都对临界资源myglobal进行操作，因此造成了混乱

下面给出更改后的运行结果

结果一（类似滴答滴答，0.0.0.交替出现）

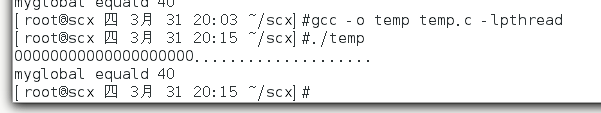


代码如下

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <errno.h>  #include <unistd.h>  pthread\_mutex\_t mutex;  int myglobal=0;  void \*thread\_function(void \*arg)  {  int i,j;  for(i = 0; i <20; i++){  pthread\_mutex\_lock(&mutex);  j = myglobal;  j++;  printf(".");  fflush(stdout);  myglobal = j;  pthread\_mutex\_unlock(&mutex);  usleep(2000);    }  return NULL;  }  int main(){  pthread\_t mythread;  pthread\_mutex\_init (&mutex, NULL);  int i;  if( pthread\_create(&mythread,NULL,thread\_function,NULL))  {  printf("error creating thread!\n");  abort();  }  for(i = 0; i <20; i++)  {  pthread\_mutex\_lock(&mutex);  myglobal++;  printf("O");  fflush(stdout);  pthread\_mutex\_unlock(&mutex);  usleep(2000);  }  if( pthread\_join(mythread,NULL))  {  printf("error joining thread!\n");  abort();  }  printf("\nmyglobal equald %d\n",myglobal);  exit(0);  } |

结果二（0000000….）

运行结果如下



代码如下

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <errno.h>  #include <unistd.h>  pthread\_mutex\_t mutex;  int myglobal=0;  void \*thread\_function(void \*arg)  {  int i,j;  for(i = 0; i <20; i++){  pthread\_mutex\_lock(&mutex);  j = myglobal;  j++;  printf(".");  fflush(stdout);  myglobal = j;  usleep(2000);  pthread\_mutex\_unlock(&mutex);      }  return NULL;  }  int main(){  pthread\_t mythread;  pthread\_mutex\_init (&mutex, NULL);  int i;  if( pthread\_create(&mythread,NULL,thread\_function,NULL))  {  printf("error creating thread!\n");  abort();  }  for(i = 0; i <20; i++)  {  pthread\_mutex\_lock(&mutex);  myglobal++;  printf("O");  fflush(stdout);  usleep(2000);  pthread\_mutex\_unlock(&mutex);    }  if( pthread\_join(mythread,NULL))  {  printf("error joining thread!\n");  abort();  }  printf("\nmyglobal equald %d\n",myglobal);  exit(0);  } |

无论是结果一还是结果二，都可以得到正确的myglobal

原理都是使用了互斥锁，临界资源不会在一个线程使用时，被另一个线程使用从而造成混乱，即保护了临界资源。