**实验5 线程的创建和管理**

**实验目的：**

1. 理解进程与线程的区别；
2. 实现单线程与多线程控制；
3. 了解POSIX线程机制，使用常用的线程操作函数；

**实验内容:**

1. 观察单线程程序和多线程程序的区别；
2. 使用常用的线程操作函数。
3. 使用线程互斥锁。

**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. **本实验涉及的系统调用**
5. 创建新线程：int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine) (void \*), void \*arg);

函数说明：

pthread\_create是类Unix操作系统（Unix、Linux、Mac OS X等）的创建线程的函数。它的功能是创建线程（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。

参数说明：

pthread\_t \*thread： 指向线程标识符的指针，返回线程ID；

const pthread\_attr\_t \*attr： 用来设置线程属性，NULL表⽰使⽤默认属性；

void \*(\*start\_routine) (void \*)： 线程运行函数的起始地址；

void \*arg： 传给线程启动函数的参数。

返回值：

成功返回0；失败返回错误码。

需要的头文件：

#include<pthread.h>

1. 线程终止：void pthread\_exit(void \*retval);

函数说明：

线程通过调用pthread\_exit函数终止执行，就如同进程在结束时调用exit函数一样。这个函数的作用是，终止调用它的线程并返回一个指向某个对象的[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88/2878304" \t "_blank)。

参数说明：

retval表示线程退出状态，通常传NULL；

1. 线程等待：int pthread\_join(pthread\_t thread, void \*\*retval);

函数说明：

pthread\_join()函数，以阻塞的方式等待thread指定的线程结束。当函数返回时，被等待线程的资源被收回。如果线程已经结束，那么该函数会立即返回。并且thread指定的线程必须是joinable的。

参数说明：

thread: 线程[标识符](https://baike.baidu.com/item/%E6%A0%87%E8%AF%86%E7%AC%A6/7105638" \t "_blank)，即线程ID，标识唯一线程；

retval: 用户定义的指针，用来存储被等待线程的返回值。

返回值 ：

0代表成功。失败，返回的则是错误号。

1. 互斥锁初始化

静态互斥锁用宏PTHREAD\_MUTEX\_INITIALIZER来初始化；动态互斥锁须在使用前用pthread\_mutex\_init来初始化。

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex,const pthread\_mutexattr\_t \*restrict attr);

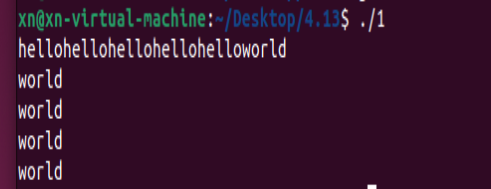
1. 锁定(获取)互斥锁（相当于P操作）：int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

当它返回时，该互斥锁已被锁定，调用它的线程就获得了这个互斥锁。如果该互斥锁已被另一个线程锁定和拥有，则调用线程将阻塞，直到该互斥锁变为可用为止。

1. 解除（释放）互斥锁锁定（相当于V操作）：int pthread\_mutex\_unlock ( pthread\_mutex\_t \*mutex );
2. **使用线程的例程**
3. 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；
2. 理解fflush( )的作用。

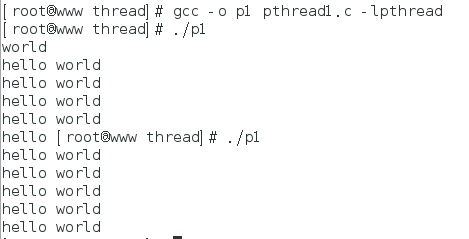
fflush()用于强制将输出缓冲区中的数据写入基础输出设备（在这种情况下是终端或控制台）

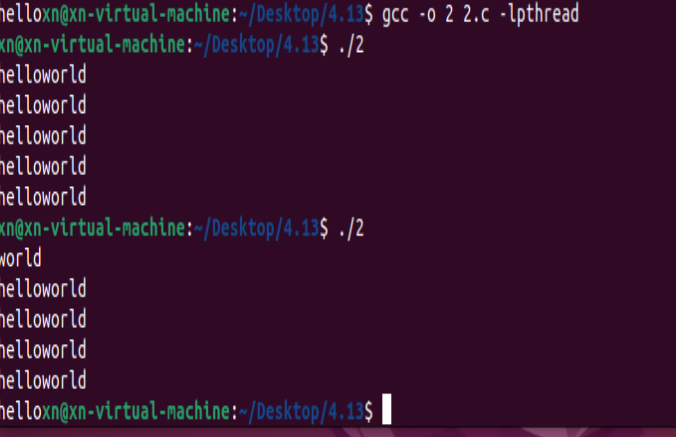
1. 如果希望一个程序能同时并发执行很多函数，它可以通过创建多个线程实现，只要将上面的程序做两个小的改动，即可让程序并发执行两个函数。下面的例程是一个简单“hello world”多线程程序。

|  |  |
| --- | --- |
| //"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;  } |

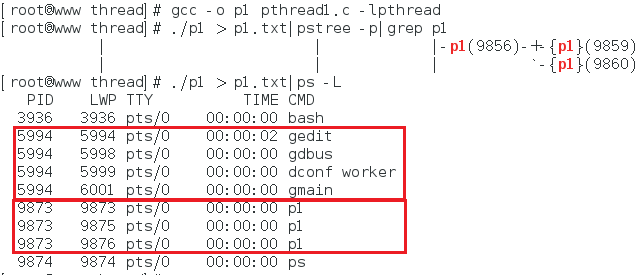
编译链接通过后，多次运行例程，观察进程和线程并发执行结果，并思考下述问题：

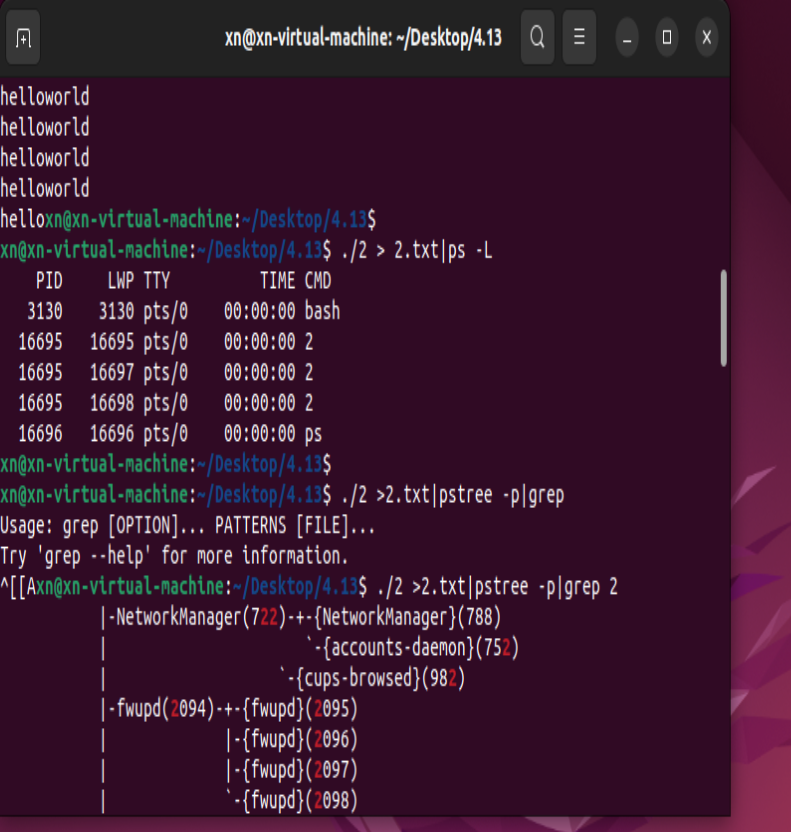
1. 从输出内容，观察线程的并发执行。





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



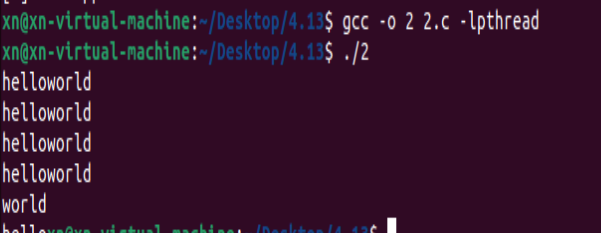


1. 理解pthread\_create( )函数的作用。

pthread\_create()函数用于创建一个新线程，该线程开始执行指定的函数。这个函数的参数包括线程标识符、线程属性、启动例程和传递给启动例程的参数。

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。

pthread\_join()函数使主线程（通常是执行main()的线程）等待指定的线程结束。如果移除这个函数调用，主线程可能会在子线程完成前结束，导致整个程序提前结束，而子线程可能没有机会完成其输出。



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。

#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>

void p\_msg(char \*m);

int main() {

pid\_t pid1, pid2;

pid1 = fork(); // 创建第一个子进程

if (pid1 == 0) { // 子进程1

p\_msg("hello");

return 0;

}

pid2 = fork(); // 创建第二个子进程

if (pid2 == 0) { // 子进程2

p\_msg("world\n");

return 0;

}

wait(NULL); // 等待任意子进程结束

wait(NULL); // 等待任意子进程结束

return 0;

}

void p\_msg(char \*m) {

for (int i = 0; i < 5; i++) {

printf("%s", m);

fflush(stdout);

sleep(1);

}

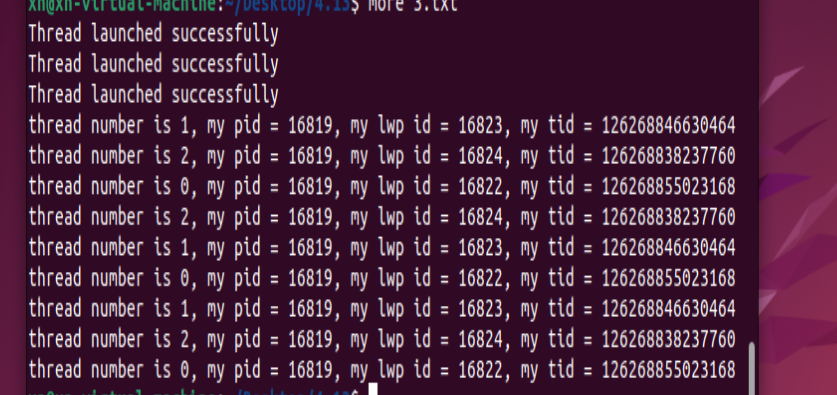
}

* 内存和资源共享：线程共享同一进程内的内存和资源，而进程拥有独立的内存空间。
* 创建和管理开销：线程的创建和管理比进程更轻量，因此通常更快，有更少的系统开销。
* 通信方式：线程可以直接通过共享内存进行通信，而进程则需要使用进程间通信机制（如管道、信号、共享内存区等）。

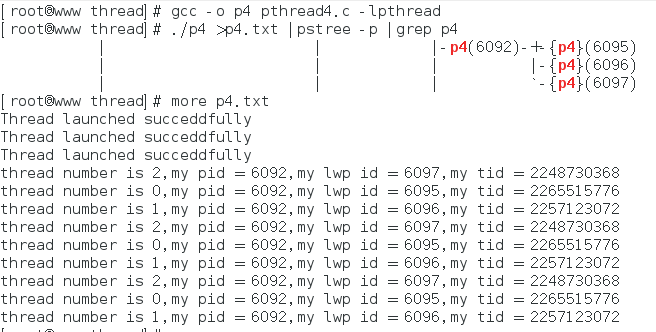
1. 运行程序，观察线程并发执行的效果，及关于线程的各种ID。

|  |
| --- |
| #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；



* 线程的调度由操作系统的线程调度器控制，通常基于多种因素（如线程优先级、CPU负载等）。
* 由于线程之间是并发执行的，每次运行程序时操作系统可能以不同的顺序调度这些线程，导致打印顺序每次可能不同。
* 线程的启动和执行速度可以受到系统当前负载和多任务处理的影响。

1. 删除sleep(1);观察输出内容的变化，并分析原因；

* 移除sleep(1);后，线程不再在打印语句之间有意地暂停，这将导致线程尽可能快地连续执行和输出。
* 由于移除了延迟，输出的速度会更快，输出间的竞争也会更激烈，可能导致输出更加混乱，甚至可能出现打印输出被中断的情况，因为多个线程可能几乎同时尝试写入到标准输出。

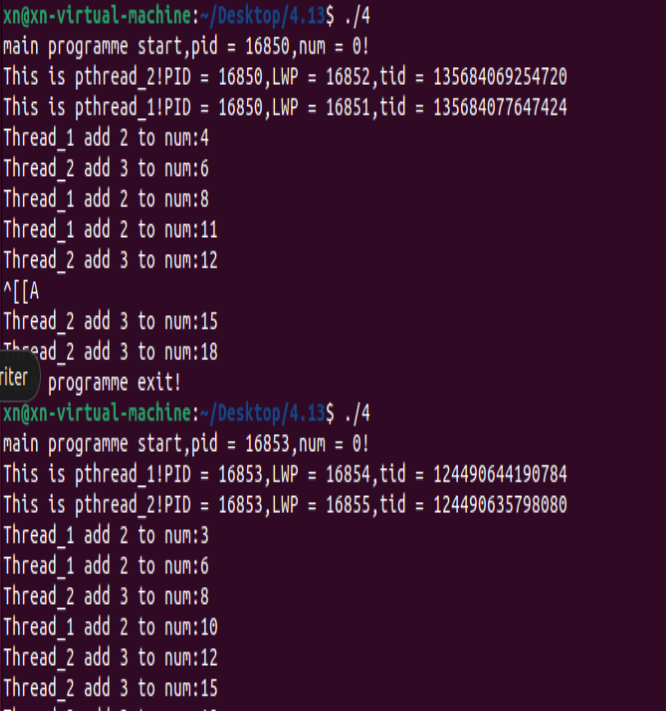
1. 分析输出内容，理解线程各种id值的含义。

* 进程ID (getpid())：返回当前进程的ID。在多线程中，所有线程共享同一进程ID。
* LWP ID (syscall(SYS\_gettid))：每个线程在内核中都有一个唯一的轻量级进程（Light-Weight Process）ID，也称为线程ID，这是操作系统内部用于区分不同线程的标识。
* 线程ID (pthread\_self())：这是由线程库提供的一个高级标识，用于在应用程序中标识不同的线程，通常是一个指针或长整型数值，具有库级别的唯一性。

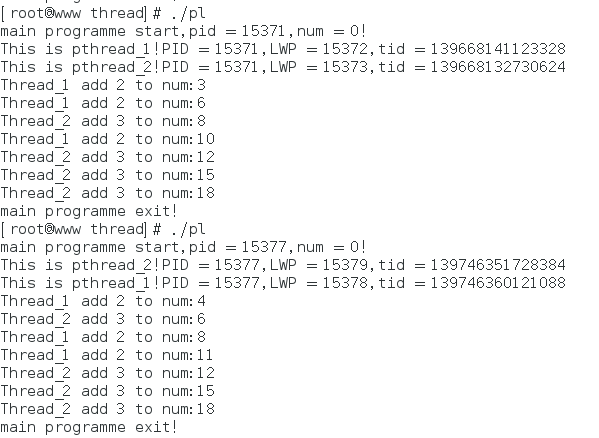
1. 运行下述例程，理解POSIX互斥锁的简单使用。

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  #include <pthread.h>  #include <errno.h>  #include <sys/types.h>  #include <sys/syscall.h>  #include <unistd.h>  #include <linux/unistd.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。

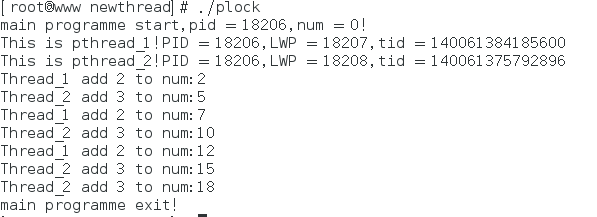


在程序中，如果保留互斥锁的相关操作注释（即不使用互斥锁），多个线程会同时访问和修改全局变量gnum。由于访问不是同步的，多个线程可能在另一个线程完成修改之前读取或再次修改该变量，导致不可预测的结果和输出混乱。

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？

移除sleep可能减少输出混乱，因为线程不会在修改全局变量后停顿，这减少了上下文切换的机会，从而可能看似减少了竞争的现象。但这并没有从根本上解决问题，因为线程仍然可能同时访问全局变量，特别是在多核处理器上，多个线程可能真正并行执行。

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；



取消注释并使用互斥锁时，互斥锁保证在任何时刻只有一个线程可以进入其保护的代码段。这避免了同时访问和修改gnum的问题，从而消除了输出混乱。

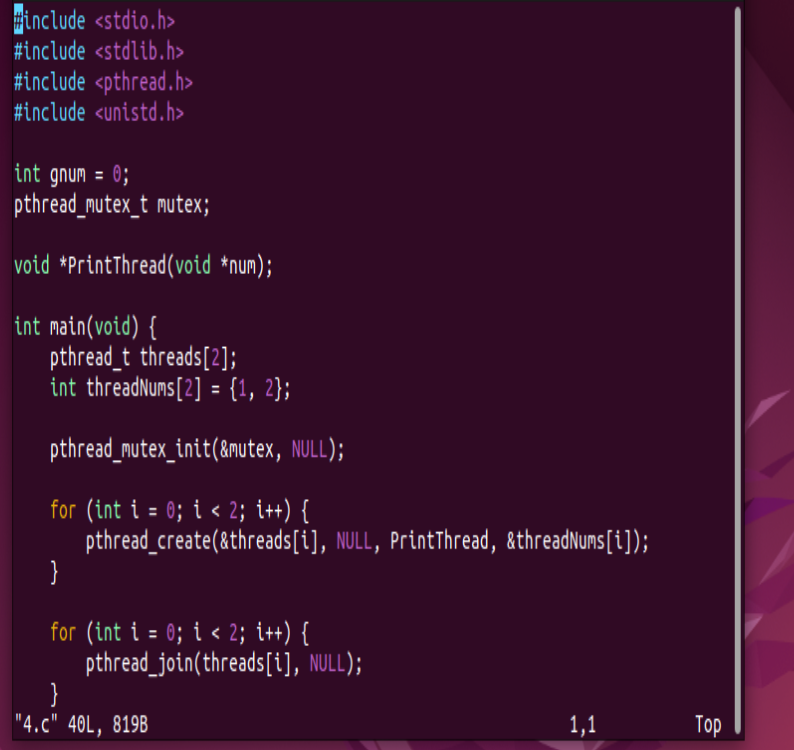
1. 掌握互斥锁的基本操作：初始化，加锁（P操作）和解锁（V操作）；

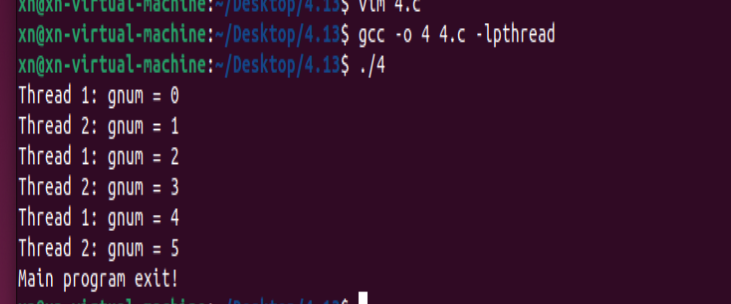
* 初始化：使用pthread\_mutex\_init(&mutex, NULL);初始化互斥锁。
* 加锁（P操作）：使用pthread\_mutex\_lock(&mutex);获取互斥锁。
* 解锁（V操作）：使用pthread\_mutex\_unlock(&mutex);释放互斥锁。

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。

虽然两种方法都减少了输出混乱，但原因不同。删除sleep减少了线程间切换的频率，从而减少了竞态条件发生的机会。而使用互斥锁则是强制性地阻止了同时访问，即使在多核处理器上也能确保数据的一致性和序列化访问。

1. 修改例程3，使用互斥锁，使各线程的输出内容不交错。

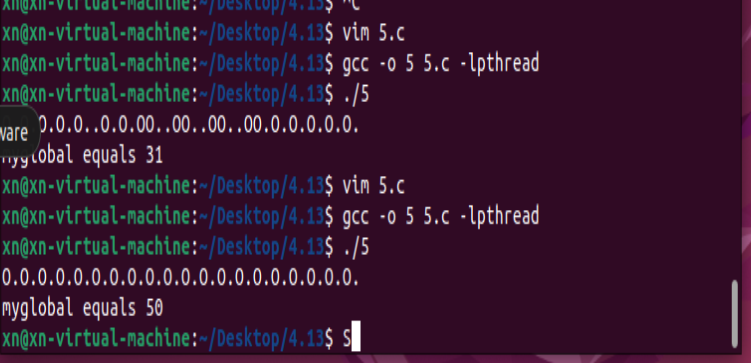


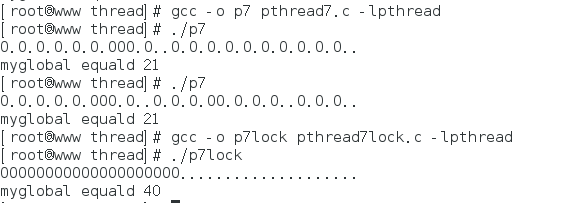


1. 练习题：运行教材习题4.9中的程序，解释原程序未获得期望值的原因。模仿例程4，使用POSIX互斥锁，保证运行结果获得期望值。

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <errno.h>  #include <unistd.h>  #include <linux/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





加锁前：不使用互斥锁时，输出中的点和大写字母'O'可能会交替出现，但由于myglobal被两个线程同时访问并修改，最终的值可能不是预期

加锁后：使用互斥锁后，每次修改myglobal前都会先获取锁，这保证了在任一时刻只有一个线程可以修改myglobal。这避免了数据竞争和竞态条件，确保每次操作都按预期进行，