1线程为什么要同步

A:共享资源，多个线程都可对共享资源操作。

B:线程操作共享资源的先后顺序不确定。

C:处理器对存储器的操作一般不是原子操作。

2互斥量

mutex操作原语

pthread\_mutex\_t

pthread\_mutex\_init

pthread\_mutex\_destroy

pthread\_mutex\_lock

pthread\_mutex\_trylock

pthread\_mutex\_unlock

3 临界区（Critical Section）

保证在某一时刻只有一个线程能访问数据的简便办法。在任意时刻只允许一个线程对共

享资源进行访问。如果有多个线程试图同时访问临界区，那么在有一个线程进入后其他所有试图访问此临界区的线程将被挂起，并一直持续到进入临界区的线程离开。临界区在被释

放后，其他线程可以继续抢占，并以此达到用原子方式操作共享资源的目的。

4 临界区的选定

临界区的选定应尽可能小，如果选定太大会影响程序的并行处理性能。

5 互斥量实例

依赖的头文件

#include<pthread.h>

函数声明

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

|  |  |
| --- | --- |
| 名称： | pthread\_mutex\_destroy |
| 功能： | 释放对互斥变量分配的资源 |
| 头文件： | #include <pthread.h> |
| 函数原形： | int  pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex); |
| 参数： |  |
| 返回值： | 若成功则返回0，否则返回错误编号。 |

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

|  |  |
| --- | --- |
| 名称： | pthread\_mutexattr\_init |
| 功能： | 初始化互斥锁。 |
| 头文件： | #include <pthread.h> |
| 函数原形： | int pthread\_mutex\_init(pthread\_mutex\_t \* mutex,  const pthread\_mutex\_t \*attr); |
| 参数： | mutex  互斥量  attr     互斥锁属性 |
| 返回值： | 若成功则返回0，否则返回错误编号。 |

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

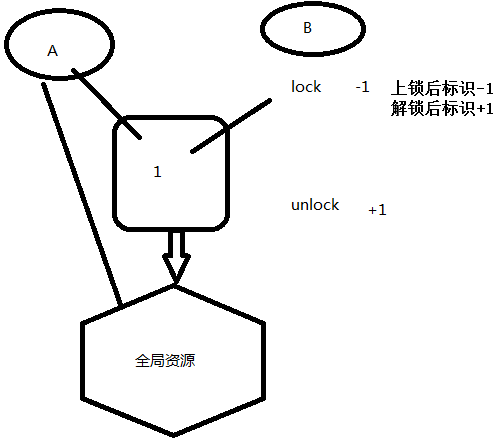
int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

|  |  |
| --- | --- |
| 名称： | pthread\_mutex\_lock/ pthread\_mutex\_trylock/ pthread\_mutex\_unlock |
| 功能： | 对互斥量加/减锁 |
| 头文件： | #include <pthread.h> |
| 函数原形： | int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);  int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);  int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex); |
| 参数： |  |
| 返回值： | 若成功则返回0，否则返回错误编号。 |

函数说明：

对互斥量进行加锁，需要调用pthread\_mutex\_lock,如果互斥量已经上锁，调用线程阻塞直至互斥量解锁。对互斥量解锁，需要调用pthread\_mutex\_unlock.

       如果线程不希望被阻塞，他可以使用pthread\_mutex\_trylock尝试对互斥量进行加锁。如果调用pthread\_mutex\_trylock时互斥量处于未锁住状态，那么pthread\_mutex\_trylock将锁住互斥量，否则就会失败，不能锁住互斥量，而返回EBUSY。



6 互斥锁创建

有两种方法创建互斥锁，静态方式和动态方式。

A：POSIX定义了一个宏PTHREAD\_MUTEX\_INITIALIZER来静态初始化互斥锁，方法如下：

pthread\_mutex\_t mutex=PTHREAD\_MUTEX\_INITIALIZER;

B:动态方式是采用pthread\_mutex\_init()函数来初始化互斥锁，API定义如下：

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

其中mutexattr用于指定互斥锁属性，如果为NULL则使用缺省属性。

C: pthread\_mutex\_destroy ()用于注销一个互斥锁，API定义如下：

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

D:pthread\_mutex\_lock()加锁

E:pthread\_mutex\_unlock()解锁

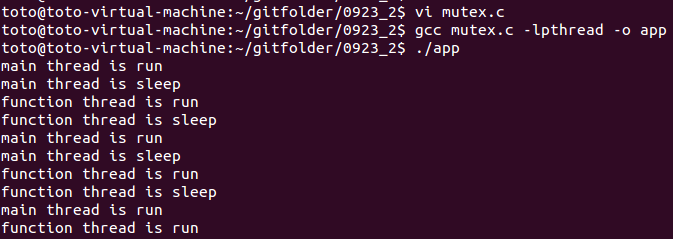
F:pthread\_mutex\_trylock()测试加锁

G:释放内存前需要调用pthread\_mutex\_destory.

案例说明互斥量加锁的必要性：

|  |
| --- |
| #include<stdio.h>  #include<pthread.h>  #include<stdlib.h>  #include<unistd.h>  void \*thread\_function(void \*arg);  /\*用run\_now代表共享资源\*/  int run\_now = 1;  int main(void)  {  int print\_count1 = 0;/\*用于循环控制\*/  pthread\_t a\_thread;  /\*创建一个进程\*/  if(pthread\_create(&a\_thread,NULL,thread\_function,NULL)){  perror("Thread creation failed!");  exit(1);  }  while(print\_count1++ < 5){  /\*主线程：如果run\_now为1就把它修改为2\*/  if(run\_now == 1) {  printf("main thread is run\n");  run\_now = 2;  } else {  printf("main thread is sleep\n");  sleep(1);  }  }  //等待子线程结束  pthread\_join(a\_thread,NULL);  exit(0);  }  void \*thread\_function(void \*arg) {  int print\_count2 = 0;  while(print\_count2++ < 5){  if(run\_now == 2) /\*子线程：如果run\_now为1就把它修改为1\*/  {  printf("function thread is run\n");  run\_now = 1;  }  else  {  printf("function thread is sleep\n");  sleep(1);  }  }  pthread\_exit(NULL);  } |

运行结果：

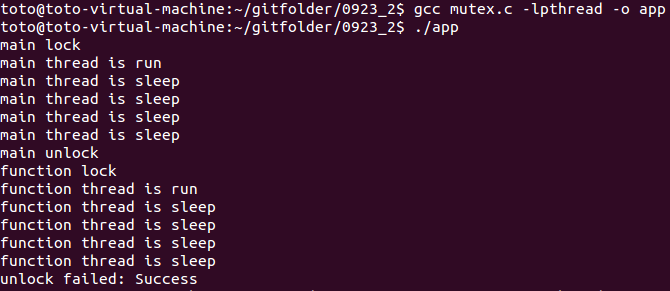


现象：main线程和function线程是交替运行的。它们都可以对run\_now进行操作。

加锁后的代码

|  |
| --- |
| #include <stdio.h>  #include <pthread.h>  #include <stdio.h>  #include <stdlib.h>  void \*thread\_function(void \*arg);  int run\_now=1; /\*用run\_now代表共享资源\*/  pthread\_mutex\_t work\_mutex; /\*定义互斥量\*/  int main(void) {  int res;  int print\_count1=0;  pthread\_t a\_thread;  if(pthread\_mutex\_init(&work\_mutex,NULL)!=0) /\*初始化互斥量\*/  {  perror("Mutex init faied");  exit(1);  }  if(pthread\_create(&a\_thread,NULL,thread\_function,NULL)!=0) /\*创建新线程\*/  {  perror("Thread createion failed");  exit(1);  }  if(pthread\_mutex\_lock(&work\_mutex)!=0) /\*对互斥量加锁\*/  {  perror("Lock failed");  exit(1);  } else {  printf("main lock\n");  }  while(print\_count1++<5) {  if(run\_now == 1) /\*主线程：如果run\_now为1就把它修改为2\*/  {  printf("main thread is run\n");  run\_now=2;  } else {  printf("main thread is sleep\n");  sleep(1);  }  }  if(pthread\_mutex\_unlock(&work\_mutex)!=0) /\*对互斥量解锁\*/  {  perror("unlock failed");  exit(1);  } else {  printf("main unlock\n");  }  pthread\_mutex\_destroy(&work\_mutex); /\*收回互斥量资源\*/  pthread\_join(a\_thread,NULL); /\*等待子线程结束\*/  exit(0);  }  void \*thread\_function(void \*arg) {  int print\_count2=0;  sleep(1);  if(pthread\_mutex\_lock(&work\_mutex)!=0) {  perror("Lock failed");  exit(1);  } else {  printf("function lock\n");  }    while(print\_count2++<5) {  if(run\_now==2) /\*分进程：如果run\_now为1就把它修改为1\*/  {  printf("function thread is run\n");  run\_now=1;  } else {  printf("function thread is sleep\n");  sleep(1);  }  }  if(pthread\_mutex\_unlock(&work\_mutex)!=0) /\*对互斥量解锁\*/  {  perror("unlock failed");  exit(1);  } else {  printf("function unlock\n");  }  pthread\_exit(NULL);  } |

运行结果如下：

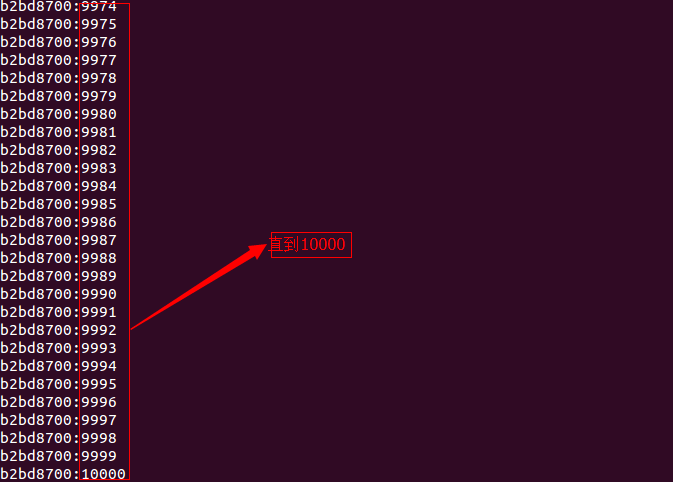


总结：从运行结果可以看到，当主进程把互斥量锁住后，子进程就不能对共享资源进行操作了，只能是同步的操作了。

|  |
| --- |
| #include<stdio.h>  #include<stdlib.h>  #include<pthread.h>  #define NLOOP 5000  int counter;  pthread\_mutex\_t counter\_mutex = PTHREAD\_MUTEX\_INITIALIZER;  void \*doit(void \*);  int main(int argc,char \*\*argv){  pthread\_t tidA,tidB;  pthread\_create(&tidA,NULL,doit,NULL);  pthread\_create(&tidB,NULL,doit,NULL);  /\*wait for both thread to terminate\*/  pthread\_join(tidA,NULL);  pthread\_join(tidB,NULL);  return 0;  }  void \*doit(void \*vptr){  int i,val;  for(i = 0;i < NLOOP;i++) {  pthread\_mutex\_lock(&counter\_mutex);  val = counter;  printf("%x:%d\n",(unsigned int) pthread\_self(),val + 1);  counter = val + 1;  pthread\_mutex\_unlock(&counter\_mutex);  }  return NULL;  } |

运行结果：





1. 死锁

A同一个线程在拥有A锁的情况下再次请求获得A锁

B线程一拥有A锁，请求获得B锁；线程二拥有B锁，请求获得A锁，出现死锁，最终导致的结果是互相等待。

