中山大學

SUN YAT-SEN UNIVERSITY

# 实 验 报 告

课程名称：操作系统

姓　　名：方桂安

学　　号：**20354027**

专业班级：**2020** 级智能科学与技术

任课教师：吴贺俊

2022 年 11 月 12 日

# 实验报告成绩评定表

| 评定项目 | 内　　容 | 满 分 | 评　分 | 总　分 |
|---|---|---|---|---|
| 实验态度 | 态度端正、遵守纪律、出勤情况 | 10 | | |
| 实验过程 | 按要求完成算法设计、代码书写、注释清晰、运行结果正确 | 30 | | |
| 实验记录 | 展示讲解清楚、任务解决良好、实验结果准确 | 20 | | |
| 报告撰写 | 报告书写规范、内容条理清楚、表达准确规范、上交及时、无抄袭，抄袭记 0 分，提供报告供抄袭者扣分。 | 40 | | |
| **评语：**<br><br>**指导老师签字：**　　年　　月　　日 | | | | |

实验报告正文

# 实验五　Linux 进程死锁

## 一、　实验目的

1. 掌握进程死锁的概念以及四个必要条件；
2. 熟悉 Linux 互斥锁的使用。

## 二、　实验内容

### 1.　任务描述

1）完成进程死锁程序代码，并且使用 pstack 查看程序调用栈

2）死锁概念简介

死锁（DeadLock)是指两个或者两个以上的进程（线程）在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程（线程）称为死锁进程(线程)。

由于资源占用是互斥的，当某个进程提出申请后，使得有关进程（线程）在无外力协助下，永远分配不到必需的资源而无法继续进行，这就产生了一种特殊现象——死锁。

## 2. 实验说明

进程死锁必要条件：

1）**对临界资源的互斥使用(资源独占)**

一个资源每次只能给一个进程（线程）使用。比如写操作

2） **占有且等待**

进程在申请新的资源的同时，保持对原有资源的占有。

3） **不可抢占**

资源申请者不能强行从资源占有者手中夺取资源，资源只能由占有者自愿释放。

4）**循环等待**

P1 等待 P2 占有的资源，P2 等待 P3 占有的资源， … Pn 等待 P1 占有的资源，形成一个进程等待回路。

# 三、 实验记录

## 1. 实施步骤

补充完成 func1 和 func2 中互斥锁的请求和释放操作:

```
int func1()
{
    // 线程 1 先占有资源 counterA，然后 sleep，确保线程 2 占有 counterB，然后再申请资源 counterB
        pthread_mutex_lock(&mutexA);
        ++counterA;
        sleep(1);
    //cout<<"The thread1 "<<pthread_self()<<" is waiting for counterB"<<endl;

    pthread_mutex_lock(&mutexB);
        ++counterB;
```

```
    //cout<<"counterA="<<counterA<<",counterB="<<counterB<<endl;

    pthread_mutex_unlock(&mutexB);
    pthread_mutex_unlock(&mutexA);

        return counterA;
}

int func2()
{
    // 线程 2 先占有资源 counterB，然后 sleep，确保线程 1 占有 counterA，然后再申请资源 counterA
        pthread_mutex_lock(&mutexB);
        ++counterB;
        sleep(1);
    //cout<< "The thread2 "<<pthread_self()<<" is waiting for counterA"<<endl;

    pthread_mutex_lock(&mutexA);
        ++counterA;
    //cout<<"counterA="<<counterA<<",counterB="<<counterB<<endl;

    pthread_mutex_unlock(&mutexA);
    pthread_mutex_unlock(&mutexB);

        return counterB;
}
```

**使用 pstack 和 gdb 工具对死锁程序进行分析**

**pstack 在 Linux 平台上的简单介绍**

pstack 是 Linux（比如 Red Hat Linux 系统、Ubuntu Linux 系统等）下一个很有用的工具，它的功能是打印输出此进程的堆栈信息。可以输出所有线程的调用关系栈。

**gdb 在 Linux 平台上的简单介绍**

GDB 是 GNU 开源组织发布的一个强大的 UNIX 下的程序调试工具。Linux 系统中包含了 GNU 调试程序 gdb,它是一个用来调试 C 和 C++ 程序的调试器。可以使程序开发者在程序运行时观察程序的内部结构和内存的使

用情况 。

gdb 所提供的一些主要功能如下所示：

1 运行程序，设置能影响程序运行的参数和环境 ；

2 控制程序在指定的条件下停止运行；

3 当程序停止时，可以检查程序的状态；

4 当程序 crash 时，可以检查 core 文件；

5 可以修改程序的错误，并重新运行程序；

6 可以动态监视程序中变量的值；

7 可以单步执行代码，观察程序的运行状态。

gdb 程序调试的对象是可执行文件或者进程，而不是程序的源代码文件。然而，并不是所有的可执行文件都可以用 gdb 调试。如果要让产生的可执行文件可以用来调试，需在执行 g++（gcc）指令编译程序时，加上 -g 参数，指定程序在编译时包含调试信息。调试信息包含程序里的每个变量的类型和在可执行文件里的地址映射以及源代码的行号。gdb 利用这些信息使源代码和机器码相关联。

## 2. 实验记录

**编译测试程序**

```
enderfga@Enderfga-PC:~$ g++ -g deadlock.cpp -o lock -lpthread
enderfga@Enderfga-PC:~$
```

**执行并查找测试程序的进程号**

```
enderfga@Enderfga-PC:~$ g++ -g deadlock.cpp -o lock -lpthread
enderfga@Enderfga-PC:~$ ./lock
```

```
enderfga@Enderfga-PC:~$ ps -ef|grep lock
enderfga    871   354  0 15:48 pts/3    00:00:00 ./lock
enderfga    898   662  0 15:48 pts/4    00:00:00 grep --color=auto lock
enderfga@Enderfga-PC:~$
```

对死锁进程第一次执行 pstack（pstack －进程号）的输出结果

```
enderfga@Enderfga-PC:~$ ps -ef|grep lock
enderfga  2422  2053  0 15:57 pts/3    00:00:00 ./lock
enderfga  2449   662  0 15:57 pts/4    00:00:00 grep --color=auto lock
enderfga@Enderfga-PC:~$ pstack 2422
Thread 5 (Thread 0x7f3322f28700 (LWP 2426)):
#0  0x00007f332497823f in clock_nanosleep () from /lib/x86_64-linux-gnu/libc.so.6
#1  0x0000000000000000 in ?? ()
Thread 4 (Thread 0x7f3323729700 (LWP 2425)):
#0  0x00007f332497823f in clock_nanosleep () from /lib/x86_64-linux-gnu/libc.so.6
#1  0x0000000000000000 in ?? ()
Thread 3 (Thread 0x7f3323f2a700 (LWP 2424)):
#0  0x00007f3324c82170 in __lll_lock_wait () from /lib/x86_64-linux-gnu/libpthread.so.0
#1  0x00007f3324c7a0a3 in pthread_mutex_lock () from /lib/x86_64-linux-gnu/libpthread.so.0
#2  0x0000000000000000 in ?? ()
Thread 2 (Thread 0x7f332472b700 (LWP 2423)):
#0  0x00007f3324c82170 in __lll_lock_wait () from /lib/x86_64-linux-gnu/libpthread.so.0
#1  0x00007f3324c7a0a3 in pthread_mutex_lock () from /lib/x86_64-linux-gnu/libpthread.so.0
#2  0x0000000000000000 in ?? ()
Thread 1 (Thread 0x7f332472c740 (LWP 2422)):
#0  0x00007f3324c78cd7 in __pthread_clockjoin_ex () from /lib/x86_64-linux-gnu/libpthread.
so.0
#1  0x00007fff03455fc8 in ?? ()
#2  0x00007fff03455f50 in ?? ()
#3  0x00007f3324c6b880 in ?? () from /lib/x86_64-linux-gnu/libstdc++.so.6
#4  0x7a8a373b09c5bd00 in ?? ()
#5  0x0000000000000000 in ?? ()
enderfga@Enderfga-PC:~$
```

对死锁进程第二次执行 pstack（pstack －进程号）的输出结果

```
enderfga@Enderfga-PC:~$ pstack 2422
Thread 5 (Thread 0x7f3322f28700 (LWP 2426)):
#0  0x00007f332497823f in clock_nanosleep () from /lib/x86_64-linux-gnu/libc.so.6
#1  0x0000000000000000 in ?? ()
Thread 4 (Thread 0x7f3323729700 (LWP 2425)):
#0  0x00007f332497823f in clock_nanosleep () from /lib/x86_64-linux-gnu/libc.so.6
#1  0x0000000000000000 in ?? ()
Thread 3 (Thread 0x7f3323f2a700 (LWP 2424)):
#0  0x00007f3324c82170 in __lll_lock_wait () from /lib/x86_64-linux-gnu/libpthread.so.0
#1  0x00007f3324c7a0a3 in pthread_mutex_lock () from /lib/x86_64-linux-gnu/libpthread.so.0
#2  0x0000000000000000 in ?? ()
Thread 2 (Thread 0x7f332472b700 (LWP 2423)):
#0  0x00007f3324c82170 in __lll_lock_wait () from /lib/x86_64-linux-gnu/libpthread.so.0
#1  0x00007f3324c7a0a3 in pthread_mutex_lock () from /lib/x86_64-linux-gnu/libpthread.so.0
#2  0x0000000000000000 in ?? ()
Thread 1 (Thread 0x7f332472c740 (LWP 2422)):
#0  0x00007f3324c78cd7 in __pthread_clockjoin_ex () from /lib/x86_64-linux-gnu/libpthread.
so.0
#1  0x00007fff03455fc8 in ?? ()
#2  0x00007fff03455f50 in ?? ()
#3  0x00007f3324c6b880 in ?? () from /lib/x86_64-linux-gnu/libstdc++.so.6
#4  0x7a8a373b09c5bd00 in ?? ()
#5  0x0000000000000000 in ?? ()
```

然后通过 gdb attach 到死锁进程

```
enderfga@Enderfga-PC:~$ gdb attach 2422
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
attach: No such file or directory.
Attaching to process 2422
[New LWP 2423]
[New LWP 2424]
[New LWP 2425]
[New LWP 2426]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
__pthread_clockjoin_ex (threadid=139857631557376, thread_return=0x0,
    clockid=<optimized out>, abstime=<optimized out>, block=<optimized out>)
    at pthread_join_common.c:145
145     pthread_join_common.c: No such file or directory.
(gdb)
```

Gdb into thread 输出：

```
(gdb) info thread
  Id   Target Id                                  Frame
* 1    Thread 0x7fa2a2045740 (LWP 3015) "lock" __pthread_clockjoin_ex (
    threadid=140336479618816, thread_return=0x0, clockid=<optimized out>,
    abstime=<optimized out>, block=<optimized out>) at pthread_join_common.c:145
  2    Thread 0x7fa2a2044700 (LWP 3016) "lock" __lll_lock_wait (
    futex=futex@entry=0x562d7bb70080 <mutex>, private=0) at lowlevellock.c:52
  3    Thread 0x7fa2a1843700 (LWP 3017) "lock" __lll_lock_wait (
    futex=futex@entry=0x562d7bb70040 <mutexA>, private=0) at lowlevellock.c:52
  4    Thread 0x7fa2a1042700 (LWP 3018) "lock" 0x00007fa2a229123f in __GI___clock_nanoslee
p (clock_id=clock_id@entry=0, flags=flags@entry=0, req=req@entry=0x7fa2a1041e10,
    rem=rem@entry=0x7fa2a1041e10) at ../sysdeps/unix/sysv/linux/clock_nanosleep.c:78
  5    Thread 0x7fa2a0841700 (LWP 3019) "lock" 0x00007fa2a229123f in __GI___clock_nanoslee
p (clock_id=clock_id@entry=0, flags=flags@entry=0, req=req@entry=0x7fa2a0840e10,
    rem=rem@entry=0x7fa2a0840e10) at ../sysdeps/unix/sysv/linux/clock_nanosleep.c:78
(gdb)
```

切换到线程 2 的输出：

```
(gdb) thread 2
[Switching to thread 2 (Thread 0x7fa2a2044700 (LWP 3016))]
#0  __lll_lock_wait (futex=futex@entry=0x562d7bb70080 <mutex>, private=0)
    at lowlevellock.c:52
52      lowlevellock.c: No such file or directory.
(gdb) where
#0  __lll_lock_wait (futex=futex@entry=0x562d7bb70080 <mutex>, private=0)
    at lowlevellock.c:52
#1  0x00007fa2a25930a3 in __GI___pthread_mutex_lock (mutex=0x562d7bb70080 <mutex>)
    at ../nptl/pthread_mutex_lock.c:80
#2  0x0000562d7bb6d2e2 in func1 () at deadlock.cpp:23
#3  0x0000562d7bb6d38e in start_routine1 (arg=0x0) at deadlock.cpp:55
#4  0x00007fa2a2590609 in start_thread (arg=<optimized out>) at pthread_create.c:477
#5  0x00007fa2a22d3133 in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:95
(gdb)
```

线程 2 和线程 3 的输出：

```
(gdb) thread 3
[Switching to thread 3 (Thread 0x7fa2a1843700 (LWP 3017))]
#0  __lll_lock_wait (futex=futex@entry=0x562d7bb70040 <mutexA>, private=0)
    at lowlevellock.c:52
52      lowlevellock.c: No such file or directory.
(gdb) f 3
#3  0x0000562d7bb6d3b6 in start_routine2 (arg=0x0) at deadlock.cpp:68
68                      int iRetValue = func2();
(gdb) p mutexA
$1 = {__data = {__lock = 2, __count = 0, __owner = 3016, __nusers = 1, __kind = 0,
    __spins = 0, __elision = 0, __list = {__prev = 0x0, __next = 0x0}},
  __size = "\002\000\000\000\000\000\000\000\310\v\000\000\001", '\000' <repeats 26 times>
, __align = 2}
(gdb) p mutexB
$2 = {__data = {__lock = 2, __count = 0, __owner = 3017, __nusers = 1, __kind = 0,
    __spins = 0, __elision = 0, __list = {__prev = 0x0, __next = 0x0}},
  __size = "\002\000\000\000\000\000\000\000\311\v\000\000\001", '\000' <repeats 26 times>
, __align = 2}
(gdb) p mutexC
$3 = {__data = {__lock = 0, __count = 0, __owner = 0, __nusers = 0, __kind = 0,
    __spins = 0, __elision = 0, __list = {__prev = 0x0, __next = 0x0}},
  __size = '\000' <repeats 39 times>, __align = 0}
(gdb)
```

## 3. 实验结果

连续多次查看这个进程的函数调用关系堆栈进行分析：当进程吊死时，多次使用 pstack 查看进程的函数调用堆栈，死锁线程将一直处于等锁的状态，对比多次的函数调用堆栈输出结果，确定哪两个线程（或者几个线程）一直没有变化且一直处于等锁的状态（可能存在两个线程 一直没有变化）。

根据上面的 pstack 输出可以发现，线程 2 和线程 3 一直处在等锁状态（pthread_mutex_lock），在连续两次的 pstack 信息输出中没有变化，所以我们可以推测线程 2 和线程 3 发生了死锁。

根据上面的 gdb 输出可以发现，线程 2 正试图获得锁 mutexB，但是锁 mutexB 已经被 LWP 为 3017 的线程得到（__owner = 3017），线程 3 正试图获得锁 mutexA，但是锁 mutexA 已经被 LWP 为 3016 的 得到（__owner = 3016），从 pstack 的输出可以发现，LWP 3016 与线程 2 是对应的，LWP 3017 与线程 3 是对应的。所以我们可以得出， 线程 2 和线程 3 发生了交叉持锁的死锁现象。查看线程的源代码发现，线程 2 和线程 3 同时使用 mutex1

和 mutex2，且申请顺序不合理。

## 四、 总结与讨论

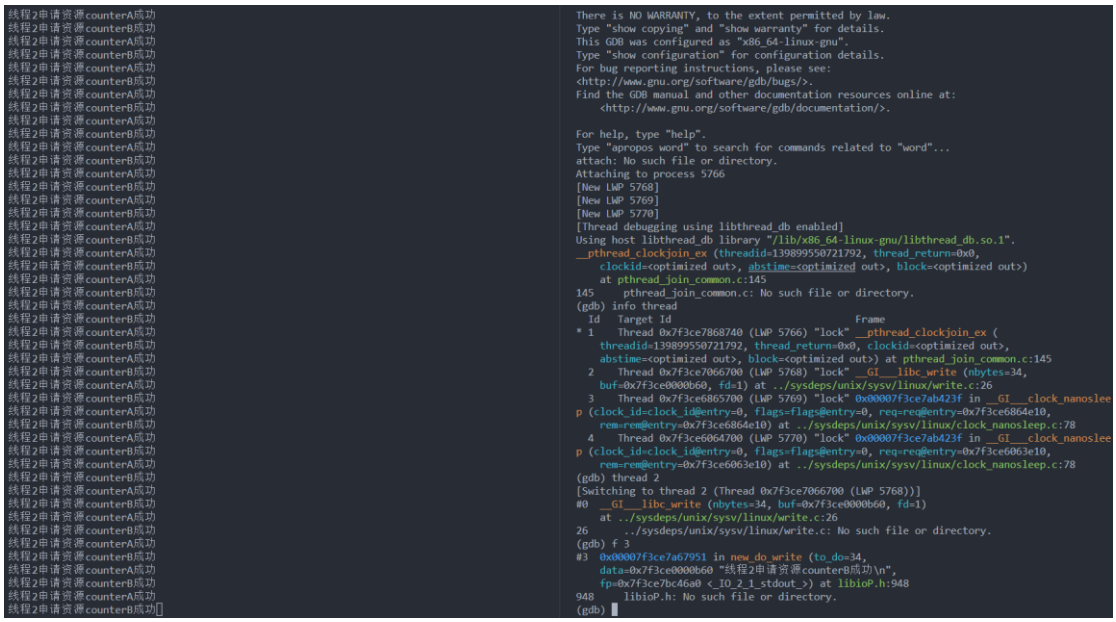
在系统设计、进程调度等方面注意如何不让这四个必要条件成立，如何确定资源的合理分配算法，避免进程永久占据系统资源。此外，也要防止进程在处于等待状态的情况下占用资源 ， 在系统运行过程中，对进程发出的每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源，若分配后系统可能发生死锁，则不予分配，否则予以分配。因此，对资源的分配要给予合理的规划，使用有序资源分配法和银行家算法等是避免死锁的有效方法。

在本次实验的代码中，一个可行的解决办法是按相同的次序锁定相应的共享资源，使用 pthread_mutex_trylock()函数，它是 pthread_mutex_lock()的非阻塞函数。

```
int func1()
{
    // 线程 1 先占有资源 counterA，然后 sleep，确保线程 2 占有 counterB，然后再申请资源 counterB
        int err;
        //pthread_mutex_lock(&mutexA);
        err=pthread_mutex_trylock(&mutexA);
        if(err==EBUSY)
        {
                cout<<"线程 1 申请资源 counterA 失败"<<endl;
                return -1;
        }
        else
        {
                cout<<"线程 1 申请资源 counterA 成功"<<endl;
        }
        ++counterA;
        //sleep(1);
    //cout<<"The thread1 "<<pthread_self()<<" is waiting for counterB"<<endl;
```

```
    //pthread_mutex_lock(&mutexB);
        err=pthread_mutex_trylock(&mutexB);
        if(err==EBUSY)
        {
                cout<<"线程 1 申请资源 counterB 失败"<<endl;
                return -1;
        }
        else
        {
                cout<<"线程 1 申请资源 counterB 成功"<<endl;
        }
        ++counterB;
    //cout<<"counterA="<<counterA<<",counterB="<<counterB<<endl;

    pthread_mutex_unlock(&mutexA);
    pthread_mutex_unlock(&mutexB);

        return counterA;
}

int func2()
{
    // 线程 2 先占有资源 counterB，然后 sleep，确保线程 1 占有 counterA，然后
再申请资源 counterA
        int err;
        //pthread_mutex_lock(&mutexB);
        err=pthread_mutex_trylock(&mutexB);
        if(err==EBUSY)
        {
                cout<<"线程 2 申请资源 counterB 失败"<<endl;
                return -1;
        }
        else
        {
                cout<<"线程 2 申请资源 counterB 成功"<<endl;
        }
        ++counterB;
        //sleep(1);
    //cout<< "The thread2 "<<pthread_self()<<" is waiting for
counterA"<<endl;

    //pthread_mutex_lock(&mutexA);
        err=pthread_mutex_trylock(&mutexA);
```

```
        if(err==EBUSY)
        {
                cout<<"线程 2 申请资源 counterA 失败"<<endl;
                return -1;
        }
        else
        {
                cout<<"线程 2 申请资源 counterA 成功"<<endl;
        }
        ++counterA;
    //cout<<"counterA="<<counterA<<",counterB="<<counterB<<endl;

    pthread_mutex_unlock(&mutexA);
    pthread_mutex_unlock(&mutexB);

        return counterB;
}
```




## 五、附：程序模块的源代码

```
#include <unistd.h>
#include <pthread.h>
#include <string.h>
#include <iostream>
using namespace std;
pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutexB = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutexC = PTHREAD_MUTEX_INITIALIZER;
```

```
static int counterA = 0;
static int counterB = 0;

// 补充完成 func1 和 func2 中互斥锁的请求和释放操作
int func1()
{
    // 线程 1 先占有资源 counterA，然后 sleep，确保线程 2 占有 counterB，然后再申请资源 counterB
        int err;
        //pthread_mutex_lock(&mutexA);
        err=pthread_mutex_trylock(&mutexA);
        if(err==EBUSY)
        {
                cout<<"线程 1 申请资源 counterA 失败"<<endl;
                return -1;
        }
        else
        {
                cout<<"线程 1 申请资源 counterA 成功"<<endl;
        }
        ++counterA;
        //sleep(1);
    //cout<<"The thread1 "<<pthread_self()<<" is waiting for counterB"<<endl;

    //pthread_mutex_lock(&mutexB);
        err=pthread_mutex_trylock(&mutexB);
        if(err==EBUSY)
        {
                cout<<"线程 1 申请资源 counterB 失败"<<endl;
                return -1;
        }
        else
        {
                cout<<"线程 1 申请资源 counterB 成功"<<endl;
        }
        ++counterB;
    //cout<<"counterA="<<counterA<<",counterB="<<counterB<<endl;

        //死锁部分的代码以下需要切换顺序
    pthread_mutex_unlock(&mutexA);
    pthread_mutex_unlock(&mutexB);
```

```
        return counterA;
}

int func2()
{
    // 线程 2 先占有资源 counterB，然后 sleep，确保线程 1 占有 counterA，然后再申请资源 counterA
        int err;
        //pthread_mutex_lock(&mutexB);
        err=pthread_mutex_trylock(&mutexB);
        if(err==EBUSY)
        {
                cout<<"线程 2 申请资源 counterB 失败"<<endl;
                return -1;
        }
        else
        {
                cout<<"线程 2 申请资源 counterB 成功"<<endl;
        }
        ++counterB;
        //sleep(1);
    //cout<< "The thread2 "<<pthread_self()<<" is waiting for counterA"<<endl;

    //pthread_mutex_lock(&mutexA);
        err=pthread_mutex_trylock(&mutexA);
        if(err==EBUSY)
        {
                cout<<"线程 2 申请资源 counterA 失败"<<endl;
                return -1;
        }
        else
        {
                cout<<"线程 2 申请资源 counterA 成功"<<endl;
        }
        ++counterA;
    //cout<<"counterA="<<counterA<<",counterB="<<counterB<<endl;

        //死锁部分的代码以下需要切换顺序
    pthread_mutex_unlock(&mutexA);
    pthread_mutex_unlock(&mutexB);

        return counterB;
```

```
}

void* start_routine1(void* arg)
{
        while (1)
        {
                int iRetValue = func1();

                if (iRetValue == 1)
                {
                        pthread_exit(NULL);
                }
        }
}

void* start_routine2(void* arg)
{
        while (1)
        {
                int iRetValue = func2();

                if (iRetValue == 1)
                {
                        pthread_exit(NULL);
                }
        }
}

void* start_routine(void* arg)
{
        while (1)
        {
                sleep(1);
                char szBuf[128];
                memset(szBuf, 0, sizeof(szBuf));
                strcpy(szBuf, (char*)arg);
        }
}

int main()
{
        pthread_t tid[4];
        if (pthread_create(&tid[0], NULL, &start_routine1, NULL) !=
0)
```

```
        {
                return -1;
        }
        if (pthread_create(&tid[1], NULL, &start_routine2, NULL) !=
0)
        {
                return -1;
        }
    long long arg = 11111111;
        if (pthread_create(&tid[2], NULL, &start_routine, &arg) != 0)
        {
                return -1;
        }
        if (pthread_create(&tid[3], NULL, &start_routine, &arg) != 0)
        {
                return -1;
        }

        sleep(5);
        //pthread_cancel(tid[0]);

        pthread_join(tid[0], NULL);
        pthread_join(tid[1], NULL);
        pthread_join(tid[2], NULL);
        pthread_join(tid[3], NULL);

        pthread_mutex_destroy(&mutexA);
        pthread_mutex_destroy(&mutexB);
        pthread_mutex_destroy(&mutexC);


        return 0;
}
```