**《操作系统》实验报告**

徽标, 公司名称

描述已自动生成

**题目: 死锁分析**

**学院 计算机学院（国家示范性软件学院）**

**班级 2021211301**

**学号 2021213586**

**姓名 郭栩源**

**2024年 4月**

## 目 录

第一章 实验概述 1

1.1 实验内容 1

1.1.1探索死锁发生条件 1

1.1.2编程实现多个线程竞争使用临界资源导致死锁 1

第二章 实验步骤 1

2.1 实验代码 1

2.2 使用GDB对死锁进行观察 4

2.2.1 GDB常用指令介绍 4

2.2.2使用GDB观察三组死锁实验 5

2.3 使用资源分配图描述三组实验的状态 7

第三章 总结 8

# **第一章 实验概述**

##### 1.1 实验内容

# **1.1.1 探索死锁发生条件**

以下是导致死锁的四个必要条件，只要系统发生死锁则以上四个条件至少有一个成立：

互斥条件：资源是独占的且排他使用，进程互斥使用资源，即任意时刻一个资源只能给一个进程使用，其他进程若申请一个资源，而该资源被另一进程占有时，则申请者等待直到资源被占有者释放。

持有并等待条件：进程每次申请它所需要的一部分资源，在申请新的资源的同时，继续占用已分配到的资源。

非抢占：进程所获得的资源在未使用完毕之前，不被其他进程强行剥夺，而只能由获得该资源的进程资源释放。

循环等待条件：在发生死锁时必然存在一个进程等待队列{P1,P2,…,Pn},其中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，形成一个进程等待环路，环路中每一个进程所占有的资源同时被另一个申请，也就是前一个进程占有后一个进程所深情地资源。

事实上循环等待的成立蕴含了前三个条件的成立，似乎没有必要列出。然而考虑这些条件对死锁的预防是有利的，因为可以通过破坏四个条件中的任何一个来预防死锁的发生。

# **1.1.2 编程实现多个线程竞争使用临界资源导致死锁**

具体可考虑以下三种情况：

1. 接口卡死，锁不能及时释放，导致别的线程卡死，这类不算真正意义的死锁。

2. 两个锁交叉使用导致死锁。

3. 同一个线程锁多次导致死锁。

# **第二章 实验步骤**

##### 2.1 实验代码

针对1.1.2中中的三种情况，设计程序进行观察。

Test1观察接口卡死的情况，threadProc1在获取资源g\_mutex后陷入无限循环，无法释放，导致threadProc2因无法获得资源而无法正常进行。注意，这种情况虽然会导致线程2的卡死，但并不是真正的死锁，即线程2的卡死并不是有死锁引起。

Test2观察两个锁交叉使用导致死锁的情况。在test2中，threadProc3先申请资源g\_mutex，然后进行1秒的休眠，在这1秒内，threadProc4申请了资源g\_mutex1，此后，threadProc3需要资源g\_mutex1以继续运行， threadProc4需要资源g\_mutex以继续运行，两者都互相需要对方持有的资源，而不释放自己的资源，最终导致死锁，两个线程都无法继续。

Test3观察同一个线程锁多次导致死锁的情况。在test3中，threadProc5先后两次申请资源g\_mutex，即在第一次申请后，释放g\_mutex前又进行了一次申请，此时threadProc5无法获得该资源，陷入死锁，程序无法继续。

以下是该实验的详细代码：

#include <stdio.h>

#include <stdlib.h>

#include <iostream>

#include <unistd.h>

#include <list>

#include <pthread.h>

#include <assert.h>

#include <semaphore.h>

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

pthread\_mutex\_t g\_mutex1 = PTHREAD\_MUTEX\_INITIALIZER;

void \*threadProc1(void \*) {

printf("threadProc1 enter = %d\n", getpid());

pthread\_mutex\_lock(&g\_mutex);

while(1);

pthread\_mutex\_unlock(&g\_mutex);

printf("threadProc1 unlock pid = %d\n", getpid());

}

void \*threadProc2(void \*) {

printf("threadProc2 enter = %d\n", getpid());

pthread\_mutex\_lock(&g\_mutex);

pthread\_mutex\_unlock(&g\_mutex);

printf("threadProc2 unlock pid = %d\n", getpid());

}

void fun3() {

printf("fun3 begin\n");

pthread\_mutex\_lock(&g\_mutex1);

pthread\_mutex\_unlock(&g\_mutex1);

printf("fun3 end\n");

}

void fun4() {

printf("fun4 begin\n");

pthread\_mutex\_lock(&g\_mutex);

pthread\_mutex\_unlock(&g\_mutex);

printf("fun4 end\n");

}

void \*threadProc3(void \*) {

printf("threadProc3 enter = %d\n", getpid());

pthread\_mutex\_lock(&g\_mutex);

sleep(1);

fun3();

pthread\_mutex\_unlock(&g\_mutex);

printf("threadProc3 unlock pid = %d\n", getpid());

}

void \*threadProc4(void \*) {

printf("threadProc4 enter = %d\n", getpid());

pthread\_mutex\_lock(&g\_mutex1);

fun4();

pthread\_mutex\_unlock(&g\_mutex1);

printf("threadProc4 unlock pid = %d\n", getpid());

}

void fun5() {

printf("fun5 begin\n");

pthread\_mutex\_lock(&g\_mutex);

pthread\_mutex\_unlock(&g\_mutex);

printf("fun5 end\n");

}

void \*threadProc5(void \*) {

printf("threadProc5 enter = %d\n", getpid());

pthread\_mutex\_lock(&g\_mutex);

fun5();

pthread\_mutex\_unlock(&g\_mutex);

printf("threadProc5 unlock pid = %d\n", getpid());

}

void Thread1() {

pthread\_t tid;

pthread\_create(&tid, NULL, threadProc1, NULL);

pthread\_detach(tid);

}

void Thread2() {

pthread\_t tid;

pthread\_create(&tid, NULL, threadProc2, NULL);

pthread\_detach(tid);

}

void Thread3() {

pthread\_t tid;

pthread\_create(&tid, NULL, threadProc3, NULL);

pthread\_detach(tid);

}

void Thread4() {

pthread\_t tid;

pthread\_create(&tid, NULL, threadProc4, NULL);

pthread\_detach(tid);

}

void Thread5() {

pthread\_t tid;

pthread\_create(&tid, NULL, threadProc5, NULL);

pthread\_detach(tid);

}

//回调接口卡主导致其他线程等锁

void test1() {

printf("test1 begin\n");

Thread1();

Thread2();

printf("test1 end\n");

}

//两个锁交叉使用导致死锁

void test2() {

Thread3();

Thread4();

}

//同一锁被一个线程锁多次

void test3() {

Thread5();

}

int main(int argc, char \*argv[]) {

// test1();

test2();

// test3();

while(1) {

sleep(1);

}

return 0;

}

##### 2.2 使用GDB对死锁进行观察

# **2.2.1 GDB常用指令介绍**

GDB，全称 GNU symbolic debugger，简称 GDB调试器，是 Linux 平台下最常用的一款程序调试器。

使用gdb时，通过以下命令对example.c编译，注意要加-g使编译带有调试信息：

$ gcc -g example.c -o example.x

，使用下列命令进入到gdb调试：

$ gdb example.x

在gdb调试中，常用的命令有以下几个：

$ list 缩略为 l，列出程序源码，每次列出10行，按回车重复运行上一命令；

$ run 缩略为 r，程序开始运行，在r后可以加入程序启动参数，程序运行到断点处暂停；

$ continue 缩略为 c，程序继续运行，到下一断点处暂停；

$ step 缩略为s，单步调试

$ next 缩略为 n，单步调试（不步入函数内部）

$ break 缩略为 b，在程序某一位置设置断点；

$ info break 缩略为 i b

查看断点信息；

　$ clear ：和break相反，clear用于清除断点。

　$ print ：打印变量或表达式的值。

　$ backtrace ：打印指定个数的栈帧(stack frame)。

　$ frame ：打印栈帧。

　$ info frame ：显示当前栈帧的详细信息。

　$ kill ：结束当前程序的调试。

　$ quit ：退出gdb。

# **2.2.2 使用GDB观察三组死锁实验**

Test1：

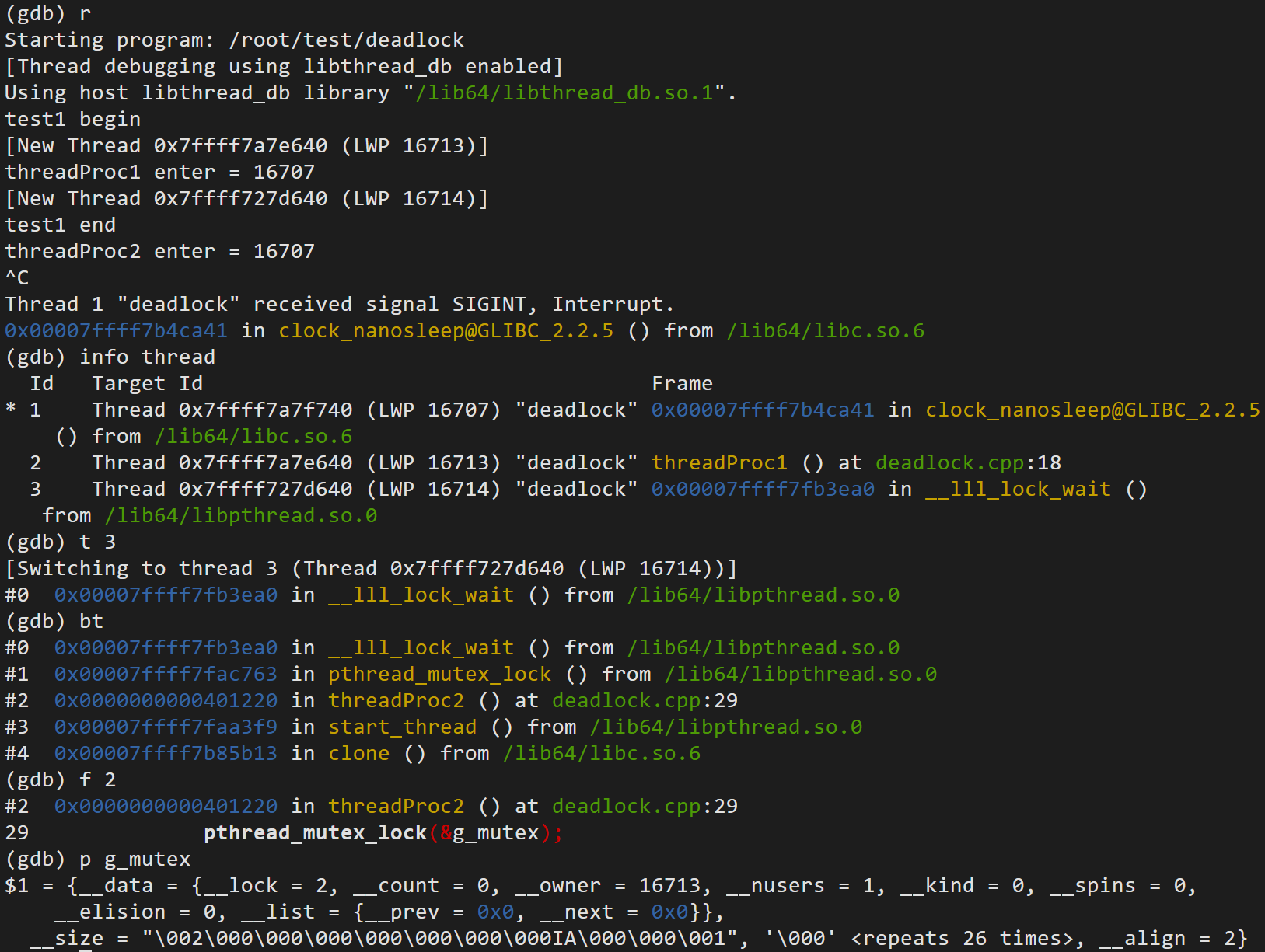
首先使用r让程序开始运行，可以看到，程序卡死（非死锁），无法退出。

使用ctrl+C中断程序，使用info thread观察当前线程，可以看到，线程二（对应Id为3）陷入了\_\_lll\_locj\_wait()。

使用t 3切换至Id为3的线程，使用bt（backtrace）显示当前线程的调用堆栈，从当前函数回溯到程序入口点pthread\_mutex\_lock()。

pthread\_mutex\_lock()由threadProc2调用，使用f 2打印栈帧，可以看到，线程二因无法获得锁g\_mutex而无法继续进行。

使用p g\_mutex打印锁g\_mutex的信息，可以看到，锁g\_mutex由LWP 16713拥有，而LWP 16713对应threadProc1。至此，我们已经可以得出结论，即线程二因为无法获得线程一占有的资源而无法继续进行。



Test2：

与实验一相同，首先使用r让程序开始运行，可以看到，程序又一次卡死，无法退出。

使用ctrl+C中断程序，使用info thread观察当前线程，可以看到，两个线程这次都陷入了\_\_lll\_locj\_wait()。

使用t 2切换至Id为2的线程（LWP 18731），使用bt（backtrace）显示当前线程的调用堆栈，从当前函数回溯到程序入口点pthread\_mutex\_lock()。

pthread\_mutex\_lock()由threadProc3调用，使用f 2打印栈帧，可以看到，线程三因无法获得锁g\_mutex1而无法继续进行。

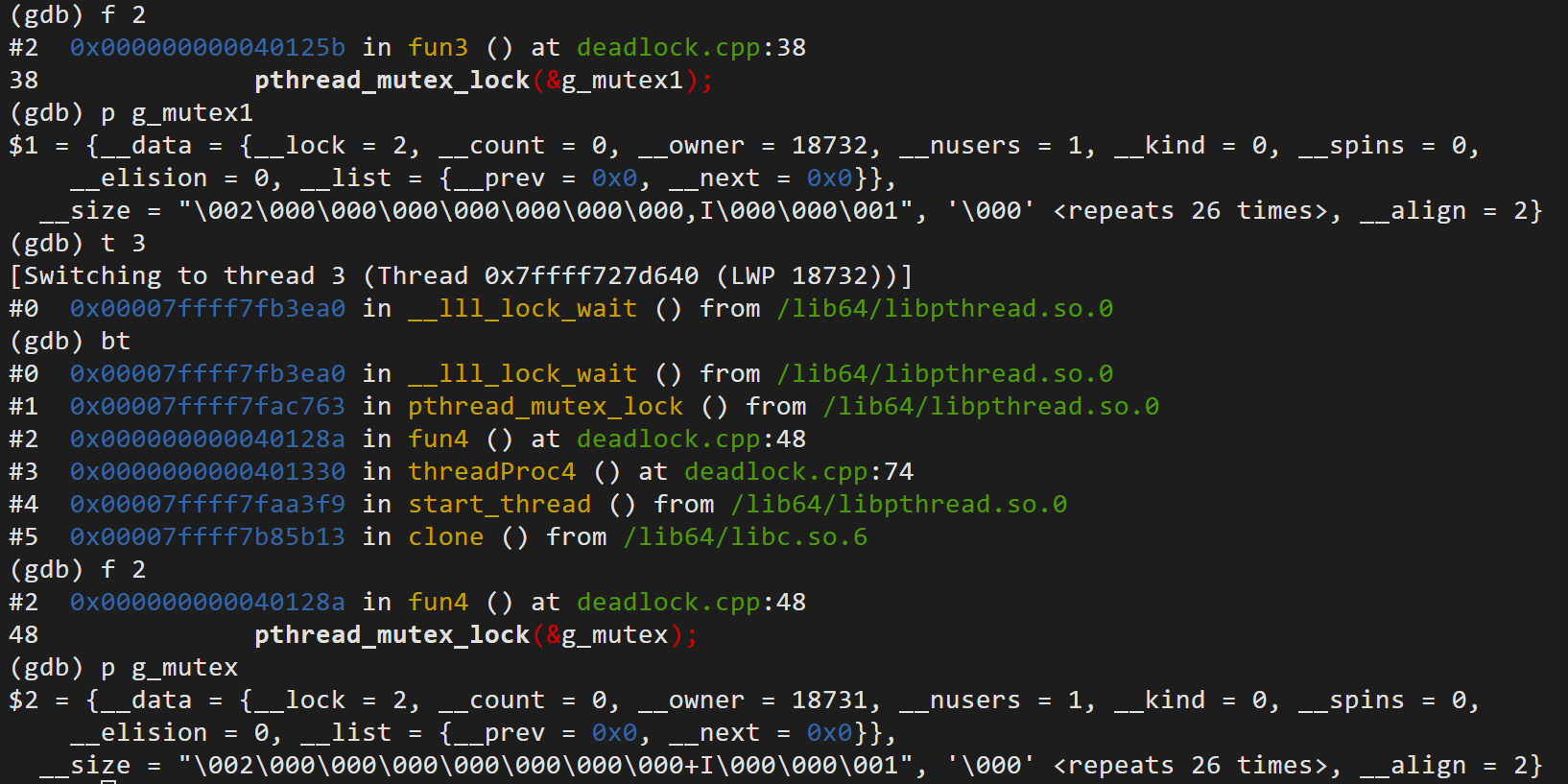
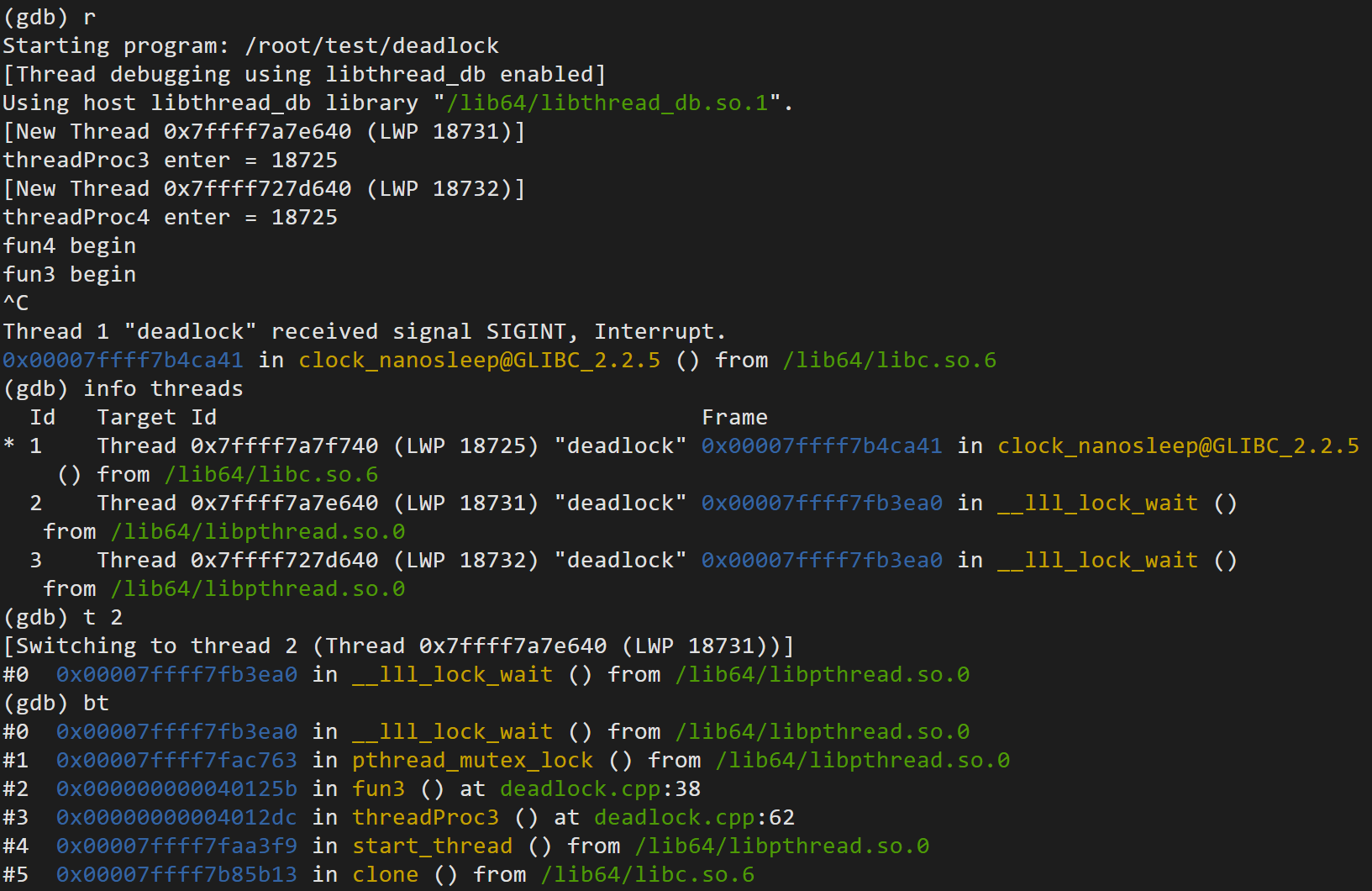
使用p g\_mutex1打印锁g\_mutex1的信息，可以看到，锁g\_mutex1由LWP 18732拥有，而LWP 18732对应threadProc4。

使用t 3切换至Id为3的线程（LWP 18732），使用bt（backtrace）显示当前线程的调用堆栈，从当前函数回溯到程序入口点pthread\_mutex\_lock()。

pthread\_mutex\_lock()由threadProc4调用，使用f 2打印栈帧，可以看到，线程三因无法获得锁g\_mutex而无法继续进行。

使用p g\_mutex打印锁g\_mutex的信息，可以看到，锁g\_mutex由LWP 18731拥有，而LWP 18731对应threadProc3。

至此，我们已经可以得出结论，即线程三和线程四彼此占有对方需要的资源，导致程序陷入死锁，无法继续进行。



Test3:

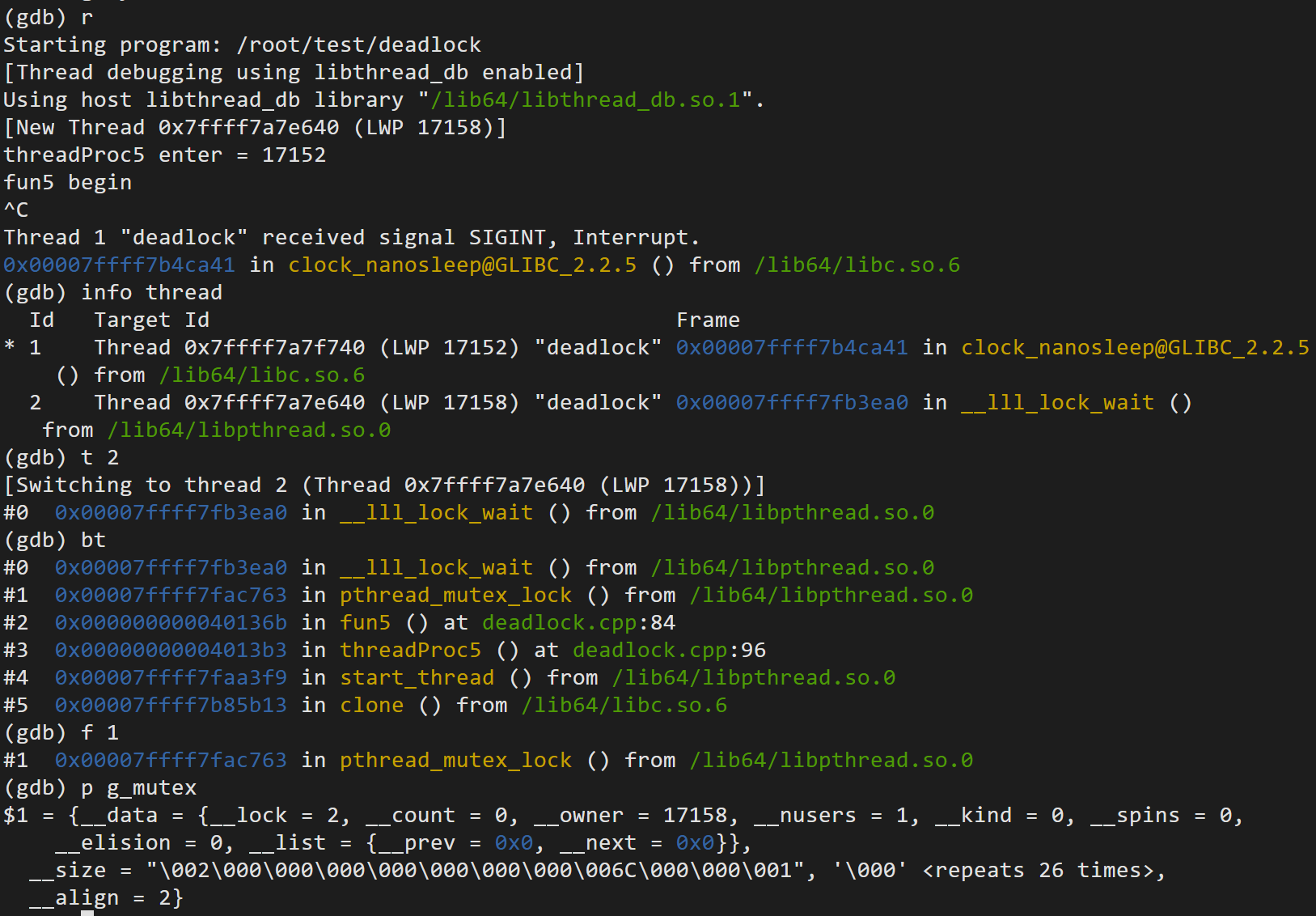
与实验一、二相同，首先使用r让程序开始运行，可以看到，程序卡死，无法退出。

使用ctrl+C中断程序，使用info thread观察当前线程，可以看到，线程五（对应Id为2）陷入了\_\_lll\_locj\_wait()。

使用t 2切换至Id为2的线程，使用bt（backtrace）显示当前线程的调用堆栈，从当前函数回溯到程序入口点pthread\_mutex\_lock()。

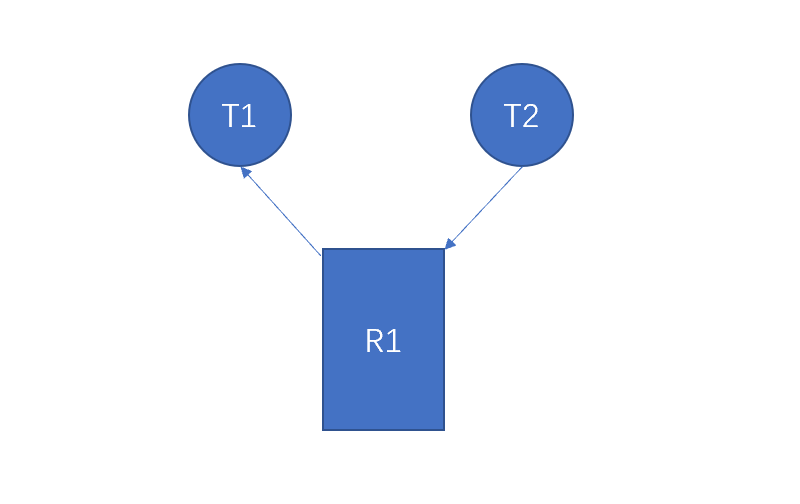
pthread\_mutex\_lock()由threadProc5调用，使用f 1打印栈帧，可以看到，线程二因无法获得锁g\_mutex而无法继续进行。

使用p g\_mutex打印锁g\_mutex的信息，可以看到，锁g\_mutex由LWP 17158拥有，而LWP 17158对应threadProc5本身。至此，我们已经可以得出结论，即线程五因为自身多次申请锁而不释放锁而陷入死锁状态，无法继续进行。



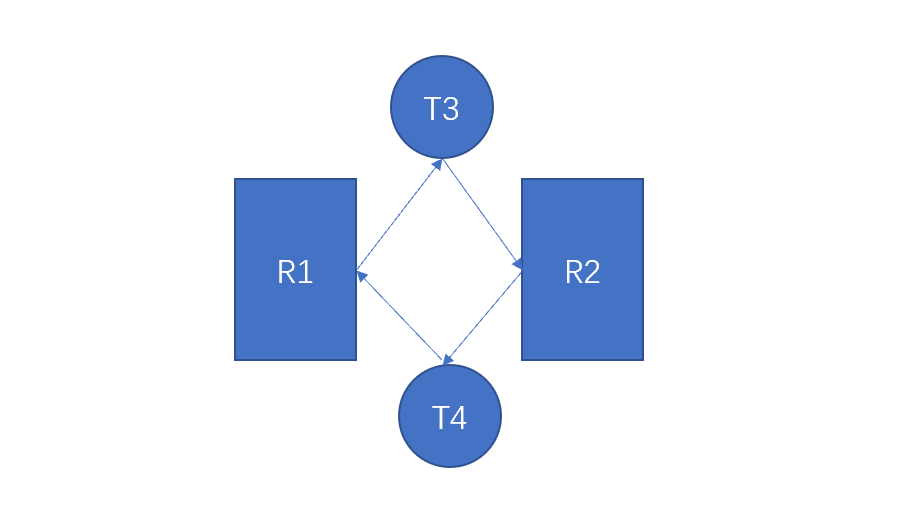
##### 2.3 使用资源分配图描述三组实验的状态

Test1为线程二需要线程一的资源，但是并不构成死锁，因此资源分配图中并不存在环路：



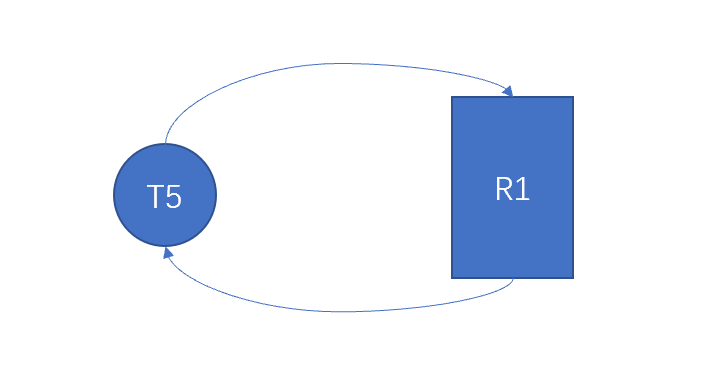
由于不存在死锁，若想要解决该种情况的卡死问题，只需要防止程序陷入死循环即可。

Test2为线程三需要线程四互相占有对方需要的资源，可以看到，资源分配图中R1、T3、R2、T4形成了环路，而环路涉及的线程T3和T4则陷入了死锁状态：



为解决该种请况的死锁问题，需要破坏环路R1、T3、R2、T4，让T3释放资源R1或T4抢占T3的资源R1即可（反过来也行）。

Test3为线程五自身多次申请锁而不释放锁而陷入死锁状态，可以看到，资源分配图中R1、T3、R2、T4形成了环路，而环路涉及的线程T3和T4则陷入了死锁状态：



在这种情况下，由于T5本身需要多个R1资源，故除了增加资源数量外没有其他解除死锁的方法。

# **第三章 实验总结**

在本次实验中，我根据引发死锁的四个必要条件：互斥条件、持有并等待条件、非抢占条件和循环等待条件，设计并实现了一个模拟程序，该程序通过多个线程尝试同时访问共享资源来展示死锁的发生。

在程序运行中，我使用gdb调试器观察程序执行中的各个细节。通过gdb，我能够跟踪每个线程的执行流程，并逐步观察到线程间如何逐渐形成死锁。调试过程中，我观察到线程在请求资源时陷入等待，直至其他线程释放锁。通过堆栈跟踪，我分析了每个线程等待的资源，从而验证了死锁发生的条件。

通过这次实验，我不仅加深了对死锁理论的理解，也掌握了使用调试工具观察和分析程序行为的实用技能。这些技能对于今后在软件开发和维护中识别和解决死锁问题将非常有用。此外，本次实验的经验也能为我们在编写更复杂多线程程序时提供宝贵的参考和指导。