厦门大学计算机科学系本科生课程

《操作系统原理》

课程实验报告

**实验一 体验Nachos下的并发程序设计**

|  |  |  |
| --- | --- | --- |
| **组 长：** | **代明亮** | |
| **小组成员：** | **代明亮** | **22920172204084** |
|  | **李笑寒** | **22920172204145** |
|  | **许惠婷** | **22920172204252** |
| **指导老师：** | **李辉、郑炜** | |
| **实验时间：** | **2020/3/6-2020/3/27** | |

目录

[实验目的 3](#_Toc36064498)

[实验内容 4](#_Toc36064499)

[实验前nachos并发编程的准备 4](#_Toc36064500)

[实现双向有序链表 4](#_Toc36064501)

[体验nachos线程系统 4](#_Toc36064502)

[实验平台 4](#_Toc36064503)

[实验过程 5](#_Toc36064504)

[实验前nachos并发编程的准备 5](#_Toc36064505)

[实验设计及关键代码 7](#_Toc36064506)

[实验结果 17](#_Toc36064507)

[非致命错误 17](#_Toc36064508)

[元素排序错误 18](#_Toc36064509)

[插入错误 18](#_Toc36064510)

[段错误 21](#_Toc36064511)

[未知错误 21](#_Toc36064512)

[实验总结 22](#_Toc36064513)

[实验分工 22](#_Toc36064514)

[错误及总结 22](#_Toc36064515)

[收获心得 24](#_Toc36064516)

# 实验目的

➢熟悉Nacho，初步体验Nachos下的并发程序设计；

➢熟悉操作系统中一些进程函数的调用；

➢熟悉和理解线程并发的概念，学习并尝试了解线程工作机制的代码；

➢进一步理解线程的工作机制，模拟没有临界保护的线程切换，分析会产生的问题和原因；

➢提高学习能力和练习编程，提高英文文档的阅读能力。

# 实验内容

## 实验前nachos并发编程的准备

➢安装Nachos；

➢尝试编译Nachos源代码；

➢阅读Nachos英文文档并阅读源代码；

➢初步尝试修改Nachos源代码：写一个最简单的程序，在运行./nachos时可以输出 “Hello Nachos”。

## 实现双向有序链表

➢自行编写三个文件dllist.h、dllist.cc和dllist-driver.cc：

‒ 给定头文件dllist.h内容提供在Nachos英文文档的3.1.1 章节；

‒ 仔细阅读Nachos英文文档的3.1.1章节并按照其描述完成；

‒ 学习gcc与g++编译器的使用方法；

➢修改nachos-3.4/code/threads/threadtest.cc和 nachos3.4/code/threads/main.cc，重新编译threads子系统， 使得链表操作可以在Nachos中运行并实现用两个线程并发操作链表。

## 体验nachos线程系统

在nachos系统中使用你所写的链表程序并演示一些并发错误。

需要做的更改有：

➢将dllist.h, dllist.cc, dllist-dirver.cc等文件拷贝到nachos-3.4/code/threads/目录中；

➢修改Makefile.common中的THREAD\_H、THREAD\_C、THREAD\_O以保证新的文件确实被编译了；

➢根据实验内容，main.cc，threadtest.cc等文件可能需要改动；

➢修改nachos3.4/code/threads/threadtest.cc ，在“适当”的位置（使得 可以观察到并发程序问题）插入 currentThread->Yield()调用以强制线程切换（注意相应文件中应包含对外变量 currentThread的声明并include头文件thread.h）， 重新编译threads子系统并运行观察出现的问题；

➢自行设计可能使并发链表操作发生错误的线程执行顺序，考虑3-5 种不同的顺序为宜。

## 实验平台

➢虚拟机 Ubuntu 18.04.4 LTS

➢服务器CentOS release 5.11 (Final) ip：59.77.8.124

# 实验过程

### 实验前nachos并发编程的准备

##### 安装nachos

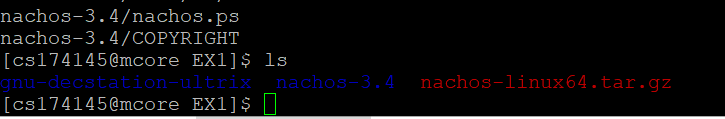
➢连接VPN，使用putty登录实验平台；

➢点击课程网站“操作系统实验nachos”文件夹下的“Nachos 源代码-适用64位Linux”，将下载的nachos-linux64.tar.gz文件拷贝到实验平台的个人目录下；

➢建立目录，解压源代码

‒ mkdir OS/EX1

‒ tar zxvf nachos-linux64.tar.gz

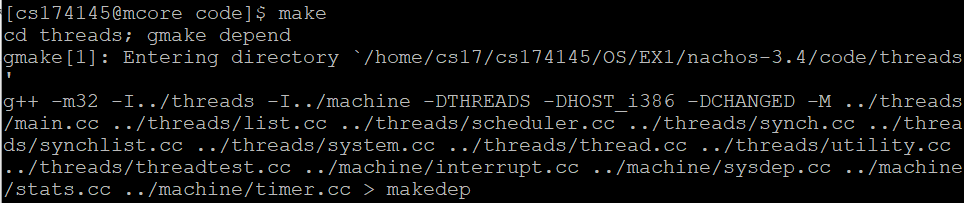


##### 尝试编译Nachos源代码

➢进入‘nachos-3.4/code’目录；

➢执行命令make；

➢观察编译结果；



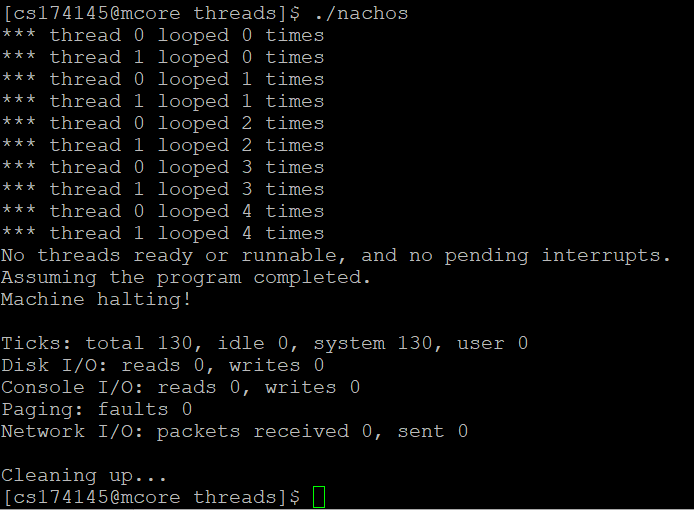
➢进入‘nachos-3.4/code/threads’目录，执行./nachos并观察结果。

‒ cd nachos-3.4/code

‒ make

‒ cd threads

‒ ./nachos



##### 阅读Nachos英文文档并阅读源代码

➢详细阅读nachos英文文档中的如下章节，并理解：

‒ 1.7 What Parts of Nachos Should We Modify Nachos.哪一部分的代码可以修改/新增，哪些代码不该被随意改动。

‒ 2.1 Installing and Building Nachos. code文件夹下的子目录有哪些文件，输入“make depend”、 “ make”、和“./nachos” 都发生了什么。

‒ 2.2 Tracing(追踪) and Debugging Nachos Programs. 了解Debug Nachos代码。

‒ 2.3 Controlling the Order of Execution in Nachos. 了解在Nachos进行context switch。

‒ 3.1.1 Priority Sorted Doubly Linked List (20 points). 提供给定头文件dllist.h内容，描述完成实验的步骤；

‒ 3.1.2 Nachos Familiarity and Understanding its Thread System (80 points). 理解ThreadTest 函数的作用。

➢详细阅读nachos-3.4/code/目录下的代码，并理解：

‒ 阅读Makefile, Makefile.dep, Makefile.common，以及nachos-3.4/code/threads/Makefile， 初步了解Makefile的构成和相互关系。

‒ 阅读threads/main.cc了解nachos如何开始。

‒ 阅读nachos-3.4/code/threads/system.cc的Initialize函数中与debug相关的部分及nachos-3.4/code/threads/utility.cc了解DEBUG的实现与使用。

‒ 阅读nachos-3.4/code/threads/threadtest.cc，了解nachos中线程的概念（用户级还是内核级线程？）及其运作方式。

##### 初步尝试修改Nachos源代码

写一个最简单的程序，比如包含一个可以输出“Hello Nachos”的Hello函数：

➢函数实现写在 hello.c里，函数声明写在hello.h里；

➢将hello的代码文件拷贝到nachos3.4/code/threads/中，相应修改nachos3.4/code/Makefile.common中的 THREAD\_H、 THREAD\_C、THREAD\_O 所在行的相关内容；

➢修改nachos-3.4/code/threads/main.cc使之包含hello.h头文件，并在main函数中调用ThreadTest 函数之前添加对Hello 函数的调用；

➢在nachos-3.4/code/threads/中依次执行“make depend” 和“ make”；

➢在nachos-3.4/code/threads/中运行“./nachos”。

‒ cd nachos-3.4/code/threads

‒ vim hello.c

#include <stdio.h>

void hellonachos()

{

printf("hello Nachos!");

return;

}

‒ vim hello.h

void hellonachos();

‒ vim main.cc

#include "hello.h"

hellonachos();

‒ cd ..

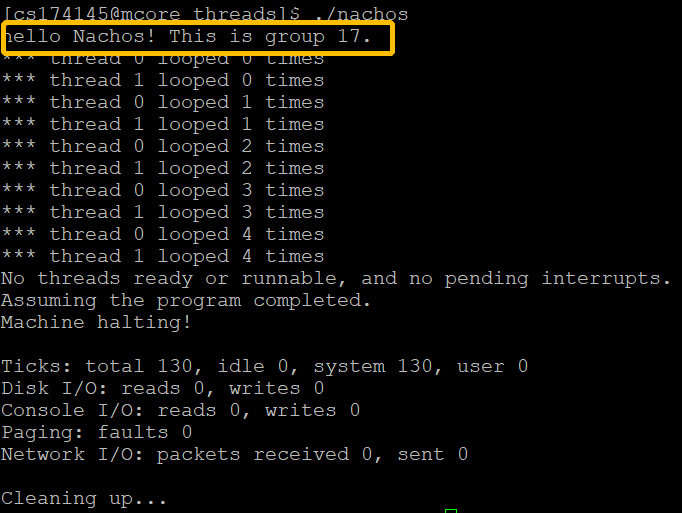
‒ vim Makefile.common

‒ cd threads

‒ make depend

‒ make

‒ ./nachos



### 实验设计及关键代码

##### 双向链表

➢建立测试目录：

为了便于单独对链表进行测试，我们单独在nachos3.4/code/中建立一个一个dllist文件夹，并将最初设计的三个文件dllist.h, dllist.cc, dllist-dirver.cc放在nachos3.4/code/dllist/中，同时，在nachos3.4/code/dllist/中还编写了简单的main.cc以及makefile文件吗，用于链表的测试。

➢dllist.h函数的简要说明以及对双向链表结构的设计：

nachos的实验说明书中关于双向链表头文件dllist.h的声明为我们实现双向链表提供了参考。dllist.h文件告诉我们了一个链表节点的基本结构：含有一个指向前一个结点的指针\*prev，一个指向后一个结点的指针\*next，一个用于结点排序的关键字key，一个结点的值\*item。在dllist.h中还进行了对链表操作的函数的声明，这些函数有：void Prepend(void \*item); void Append(void \*item); void \*Remove(int \*keyPtr); bool IsEmpty();void SortedInsert(void \*item, int sortKey);void \*SortedRemove(int sortKey); void Show();void dllFunc1(DLList \*list, int N);void dllFunc2(DLList \*list, int N);

➢设计dllist.cc中的函数：

在dllist.cc文件中，我们的只要工作是设计在dllist.h中声明的各个函数。这一步骤中，我们只是简单的实现了一个双向链表的功能，并未考虑将接下来的线程并发设计中界面的可视化输出以及线程的切换yield()函数加入到dllist.cc文件的函数中。

我们实现的函数功能有：

‒ DLLElement::DLLElement( void \*itemPtr, int sortKey)初始化一个链表元素，将prev和next都置空(NULL)，并初始化key和item的值；

‒ void DLList::Prepend(void \*item) 向链表的前端插入一个新的结点，并将这个结点的key值置为原头结点key值-1，如果原链表为空，将新结点的key值置为10；

‒ void DLList::Append(void \*item) 向链表的末端插入一个新的结点，并将这个结点的key值置为原尾结点key值+1，如果原链表为空，将新结点的key值置为10；

‒ void \*DLList::Remove(int \*keyPtr) 将链表的头结点删除，若链表为空，则报错”Remove error!”；

‒ bool DLList::IsEmpty() 判断链表是否为空；

‒ void DLList::SortedInsert(void \*item, int sortKey) 向链表中插入一个带有key值（sortKey）的新结点，并按照sortKey的值在链表中排序；

‒ void \*DLList::SortedRemove(int sortKey) 将链表中key值等于sortKey的结点删除；

‒ void DLList::Show() 将链表从头至尾打印出来。

➢在dllist-driver.cc中设计两个用于线程测试的driver function：

在dllist-driver.cc中定义了两个函数：dllFunc1和dllFunc2，来模拟双向链表的插入和删除操作。

‒ void dllFunc1(DLList \*list, int N) 功能是向list指向的链表中随机插入N个结点，并且将链表排序输出。

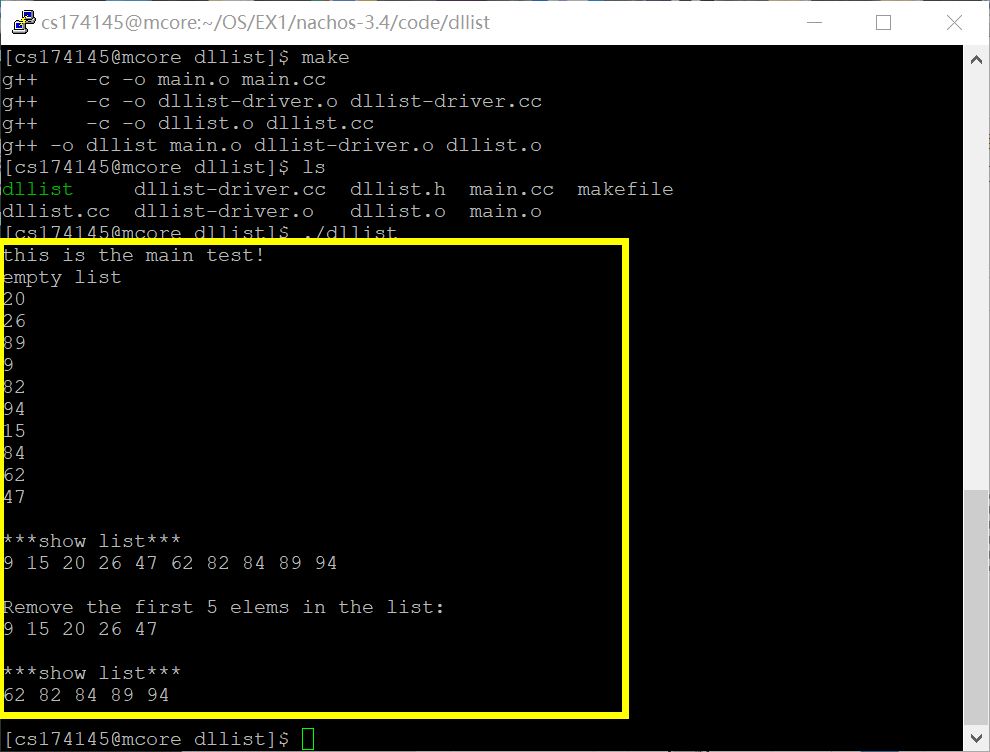
‒ void dllFunc2(DLList \*list, int N) 功能是从链表头开始连续删除N个结点，并且将链表排序输出。

➢在nachos3.4/code/dllist/中测试链表：

‒ 编写main.cc函数，使之包含dllist.h头文件，main函数的主要功能就是，先判断链表是否非空，然后在链表中通过dllFunc1(list,10);方法插入十个结点，并打印输出；通过dllFunc2(list,5);从链表头部删除5个结点，并将链表打印输出；

‒ 还需要makefile文件生成依赖文件，进行自动编译，输出可执行文件。

‒ 测试的结果为：



##### 链表测试及线程并发

命令行参数说明

**-q testnum**

-q 参数指明测试样例编号，用于演示不同的并发错误。默认值为1，即nachos自带的线程示例。该值作为全局变量可由任一线程在执行期间访问。

**-T threadnum**

-T 参数指明并发进程数，默认值为2，即有两个线程并发操作链表。注意这里的threadnum并不包括主线程（main）。后面将详细说明线程的创建方法。

**-N elementnum**

-N 参数给出了调用dllist-driver.c中两个函数（创建链表和删除链表元素）的默认参数，由于本次实验中由于要展示各种可能的错误，需要自己设计调用的过程及参数，所以该参数很少用到。

**-代码实现如下：**

```c++

switch (argv[0][1]) {

case 'q':

testnum = atoi(argv[1]);

argCount++;

break;

case 'N':

N = atoi(argv[1]);

argCount++;

break;

case 'T':

threadnum = atoi(argv[1]);

argCount++;

break;

default:

testnum = 1;

N = 10;

threadnum = 2;

break;

}```

线程的创建与使用

nachos中已经定义好了Thread类，创建时可以指定其名称。该名称最好各不相同，方便后面根据线程名做条件判断。这里的threadnum个线程通过以下的循环创建，名字分别为'0','1','2'...'threadnum-1'，\*\*不包括主线程main\*\*，也就是说main线程在创建完threadnum个线程之后不再执行任何函数，而是直接返回，剩下的就是这新创建的threadnum个线程并发执行。

这里要注意的是threadname数组需要定义为全局变量，这样才不会在main线程退出后销毁其占用的空间。以便后续通过调用currentThread->getName()获得当前进程正确的线程名。（全局变量保存在堆中。局部变量保存在栈中，函数调用结束，栈中的空间被回收。）

由于编译器是较低的版本，不支持数字转字符串的itoa函数，所以这里调用效率比较低的sprintf来实现数字到字符串的转换。

代码实现如下：

```c++

char threadname[10][5]={{0}};

for (int i=0;i<threadnum;++i)

{

sprintf(threadname[i],"%d",i);

Thread \*t = new Thread(threadname[i]);

t->Fork(SimpleThreadFunc2, 3);

}```

进程间切换（Yield）的条件判断

多个线程并发操作同一链表时，需要精心设计线程间切换的时机以捕捉偶尔会出现的并发错误。nachos中通过调用currentThread->Yield( ) 使当前进程主动放弃CPU的使用权，进入阻塞态。其中currentThread是一个全局变量，表示当前正在运行的线程。

确认了线程间切换的机制后，下面就是在线程执行过程中合适的位置插入线程切换的调用。由于实验要求一次编译即可演示多组测试样例，所以我们将当前线程名（通过调用currentThread->getName( )获得）以及测试样例编号（testnum）作为currentThread->Yield( ) 执行的判断条件，以下是代码示例：

```c++

if (testnum == 80 && strcmp(currentThread->getName(), "0") == 0)

{

printf("\n\033[1;31;40m//======= thread %s yield =======\\\\\033[m\n", currentThread->getName());

currentThread->Yield();

printf("\n\\\\------- thread %s is running -------//\n", currentThread->getName());

}

```

通过在不同的位置使用不同的条件判断来决定是否进行线程切换，实现了不同测试样例和不同线程之间执行的逻辑独立，也使团队之间的并发错误设计不会互相干扰。

##### 出错设计

这部分内容是由小组成员先集中讨论并统一代码风格，然后独立分工设计出错情况，最后统一汇总并归类和筛选出典型的几种错误；

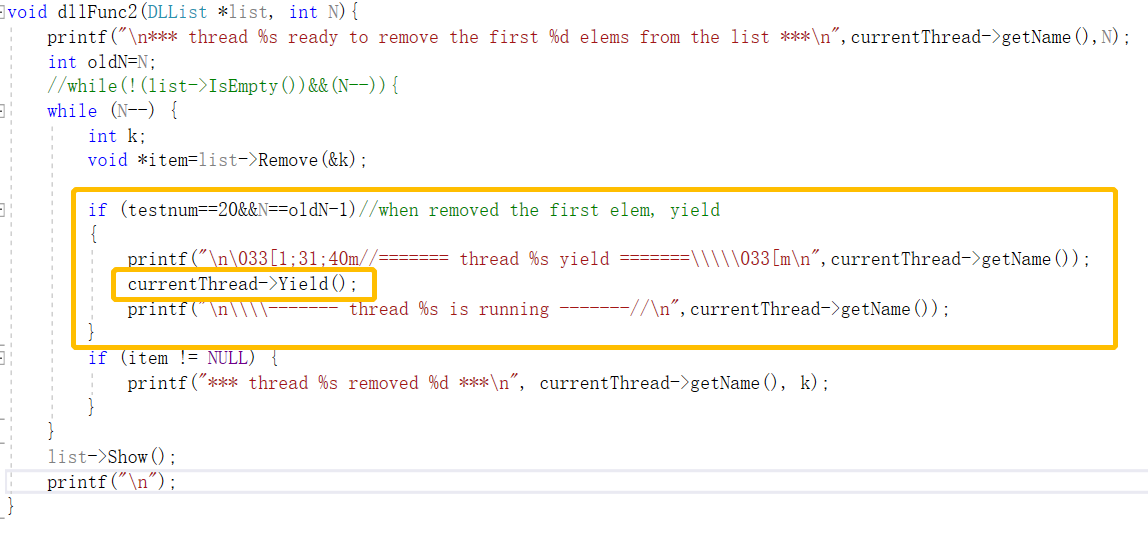
在这一步骤中，对nachos3.4/code/threads/下的dllist.cc和dllist-driver.cc以及threadtest.cc文件都进行了修改，yield()函数插入在这三个文件的不同位置，同时还对代码的可视化输出内容继续丰富；

最后，我们确定了5种错误类型，非致命错误、插入元素失序、插入错误、段错误、未知错误，每种类型中包含了不同的情况，这些错误的说明附在提交代码压缩包中error.txt文件内；

非致命错误

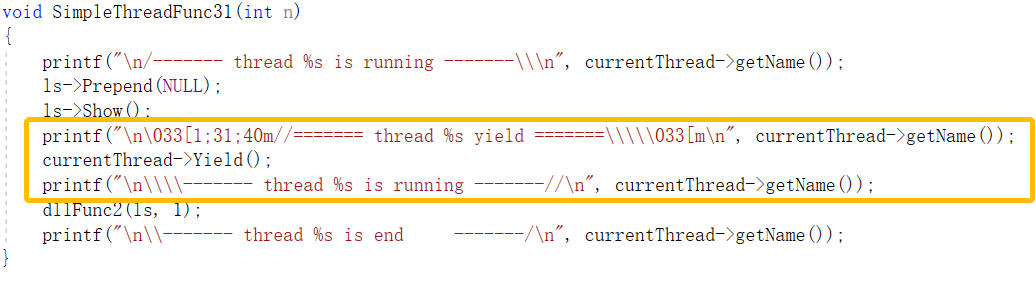
在对链表不同结点的插入或者删除操作的间隔中，发生currentThread->Yield();线程切换，造成每次对结点的删除时操作的链表结点都在被其他线程修改，导致本线程想要删除的结点已经被其他线程改变，因而造成list->Remove(&k)错误的结点或者list->Remove(&k)的结点丢失。

➢【Remove乱序】case20 testnum=20 threadnum=3三线程并发，每个线程执行的操作都是dllFunc2(ls,3);删除链表的头三个结点。但是在每一个线程list->Remove(&k)第一个结点后，使用currentThread->Yield();切换线程。由于每次list->Remove(&k)都是链表的头结点，但因为线程切换，其他线程改变了头结点，因此每个线程最后输出的值不是连续的三个，这就造成了Remove乱序。



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| threadmain | thread0 | thread1 | thread2 | list |
|  |  |  |  | NULL |
| dllFunc1 10 |  |  |  | 0 1 2 3 4 5 6 7 8 9 |
|  | Remove 0 |  |  | 1 2 3 4 5 6 7 8 9 |
|  |  | Remove 1 |  | 2 3 4 5 6 7 8 9 |
|  |  |  | Remove 2 | 3 4 5 6 7 8 9 |
|  | Remove 3 4 |  |  | 5 6 7 8 9 |
|  |  | Remove 5 6 |  | 7 8 9 |
|  |  |  | Remove 7 8 | 9 |

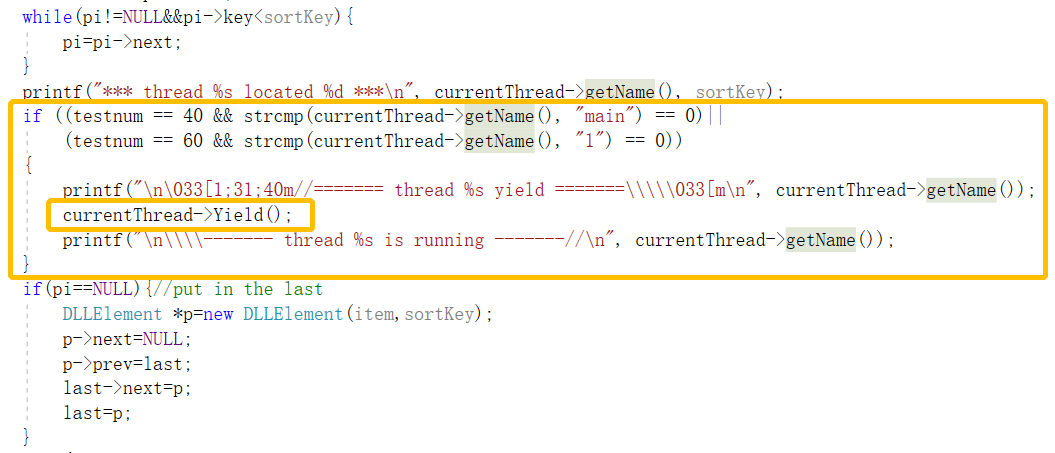
➢【Remove失败】case30 testnum=30 threadnum=2双线程并发，一个线程向空链表中使用ls->Prepend(NULL)插入一个结点，并打算dllFunc2(ls, 1)删除它之前，被另一个线程currentThread->Yield()切换，这个新结点被刚切换的进程删除，因此原线程在删除时链表已经为空，导致Remove失败。



|  |  |  |  |
| --- | --- | --- | --- |
| threadmain | thread0 | thread1 | list |
| dllFunc1 0 |  |  | NULL |
|  | Prepend 1 |  | 10 |
|  |  | dllFunc2 1 | NULL |
|  | dllFunc2 1 |  | Remove failure! |

元素排序错误

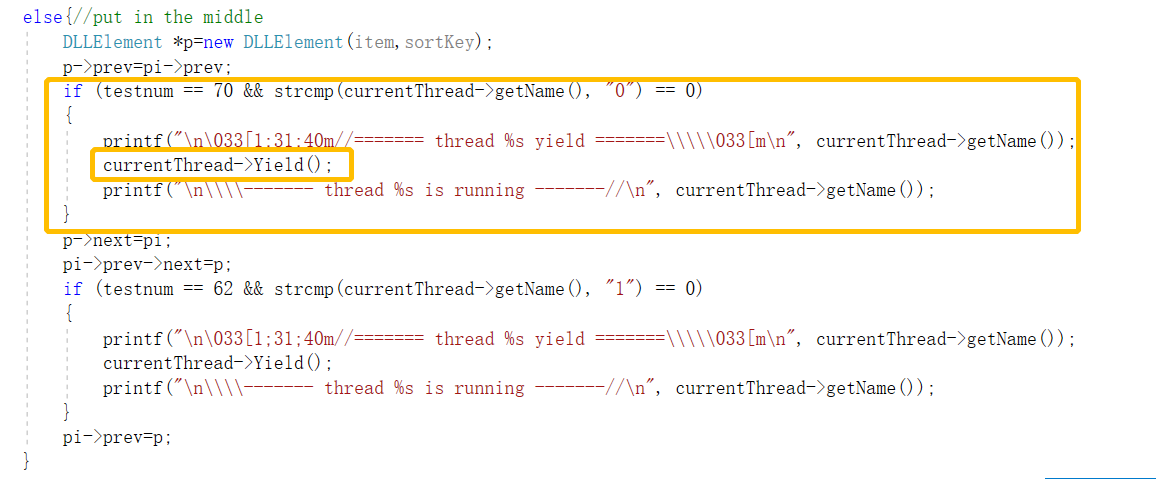
➢【乱序错误】case60 testnum=60 threadnum2双线程并发，当链表中已经有排好序的元素后，另外两个线程分别打算向链表再SortedInsert(NULL, n)插入新的元素，并且两个线程操作的是同一地址。当一个线程在SortedInsert(NULL, n)插入过程中刚刚定位好pi指针（要插入的结点位置）时切出，另一个线程开始了一个完整的插入操作，则第二个线程操作完成后，由于改变了链表位置，当第一个线程重新切入时，仍然使用初始的定位导致插入到了错误的位置上。



|  |  |  |  |
| --- | --- | --- | --- |
| threadmain | thread0 | thread1 | list |
| SortedInsert 5 15 |  |  | 5 15 |
|  | SortedInsert 11  (located 15) |  | 5 15 |
|  |  | SortedInsert 12 | 5 12 15 |
|  | SortedInsert 11 |  | 5 12 11 15 |

插入错误

➢【链表断裂】case70 testnum=70 threadnum2双线程并发，当链表中已经有排好序的元素队列时，有两个线程同时希望对链表同一地址进行操作。当一个线程0在链表中间SortedInsert(NULL,7)插入一个结点时，在修改链表指针的中途切出，另外一个线程1开始对同一地址空间的结点进行了完整的SortedRemove(7)删除操作，并修改了指针，原来线程0切入后，继续原来的指针修改，因此将重设指针，覆盖了刚刚切入的线程1对链表指针的修改，造成数据丢失，即链表断裂。



|  |  |  |  |
| --- | --- | --- | --- |
| threadmain | thread0 | thread1 | list |
| dllFunc1 10 |  |  | 0 1 2 3 4 5 6 7 8 9 |
|  | SortedInsert 7  (located 7) |  | 0 1 2 3 4 5 6 7 8 9 |
|  |  | SortedRemove 7 | 0 1 2 3 4 5 6 8 9 |
|  | SortedInsert 7 |  | 0 1 2 3 4 5 6 7 7 |

➢【插入重复】case80 testnum=80 threadnum2双线程并发，当链表中已经有元素时，有两个线程同时希望对链表尾端进行插入Append(NULL)操作。当一个线程0在链表尾端Append(NULL)插入一个结点时，在修改尾指针前切出，另外一个线程1切入并进行了完整的插入Append(NULL)操作，当线程0切回时，链表的链接关系将打乱，并形成环，导致链表重复插入两个key值相同的结点。

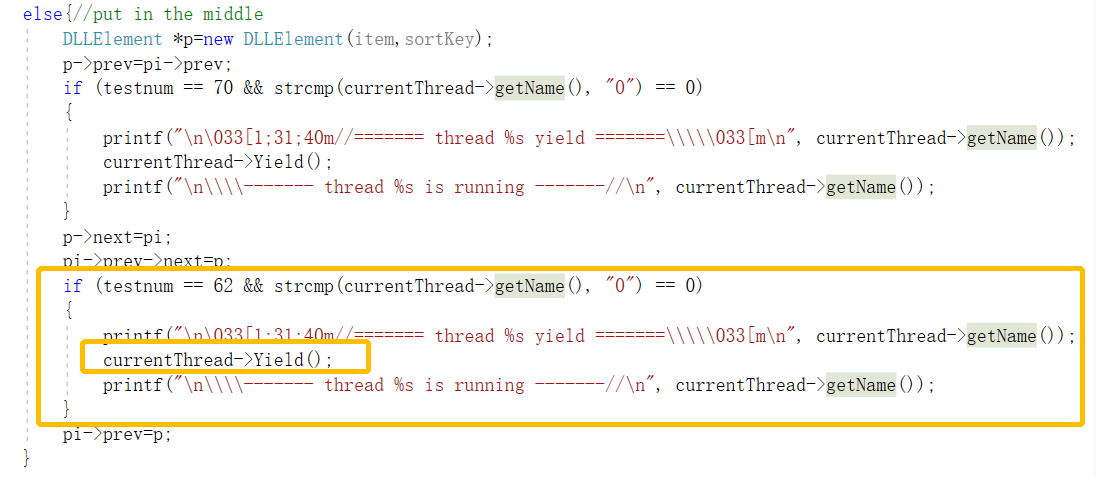


|  |  |  |  |
| --- | --- | --- | --- |
| threadmain | thread0 | thread1 | list |
| dllFunc1 3 |  |  | 0 1 2 |
|  | Append 1-yield |  | 0 1 2 |
|  |  | Append 1 | 0 1 2 3 |
|  | Append 1- continue |  | 0 1 2 3 3 |

➢【插入覆盖—链表结构错误】case81 testnum=81 threadnum2双线程并发，情况与case80类似，只不过线程0是在尾指针移动到新的尾结点前切出，线程1进行了完整的Append(NULL)插入操作后，当线程0切回时，线程1建立的链接关系同样将被线程0覆盖，导致链表结构出现错误，最终只显示插入了线程0的结点，线程1插入的结点被覆盖。

|  |  |  |  |
| --- | --- | --- | --- |
| threadmain | thread0 | thread1 | list |
| dllFunc1 3 |  |  | 0 1 2 |
|  | Append 1-yield |  | 0 1 2 |
|  |  | Append 1 | 0 1 2 3 |
|  | Append 1- continue |  | 0 1 2 3 |

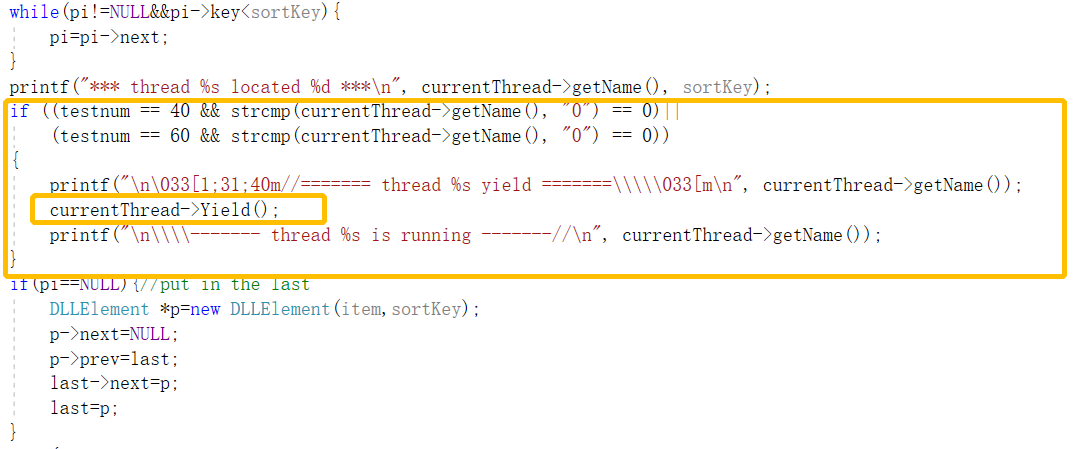
➢【插入覆盖—链表结构正常】case62 testnum=62 threadnum2双线程并发，当链表中已经有元素的时候，两个线程同时向链表同一个位置插入SortedInsert(NULL, n)不同的结点，当线程0定位后，在修改指针的过程中，被线程1切出，并且在线程1进行了完整的插入SortedInsert(NULL, n)操作后切回，这时线程0保持切出前对插入位置的判断，但是这段链表结构已经更改，因此线程0插入的元素将被线程1覆盖，因而插入失败，但是这个过程中，线程0并未破环线程1更改的链表结构，因此链表的结构正常。



|  |  |  |  |
| --- | --- | --- | --- |
| threadmain | thread0 | thread1 | list |
| SortedInsert 4 15 |  |  | 4 15 |
|  | SortedInsert 10  （located 10） |  | 4 15 |
|  |  | SortedInsert 11 | 4 11 15 |
|  | SortedInsert 10 |  | 4 11 15 |

段错误

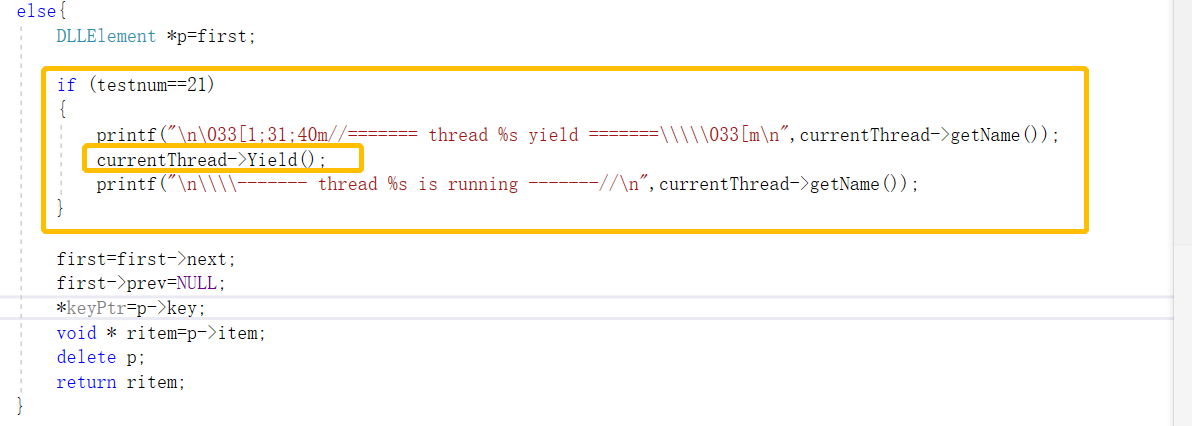
➢【段错误】case40 testnum=40 threadnum2双线程并发，两个线程同时想要对同一个地址的结点进行操作，当一个线程在定位之后，就立即被另一个线程切出，而另一个线程对这篇空间进行操作后删除了（delete p）这片空间，导致在原线程切回时，其原来定位好的指针变空，发生段错误，使得程序终止。



|  |  |  |  |
| --- | --- | --- | --- |
| threadmain | thread0 | thread1 | list |
| dllFunc 1 |  |  | 0 |
|  | SortedInsert 0  （located 0） |  | 0 |
|  |  | SortedRemove 0 | NULL |
|  | SortedInsert 0 |  | Segmentation fault |

未知错误

➢【错误“已放弃”】case21 testnum=21 threadnum3三线程并发，当链表内有足够多的元素共删除的时候，三个线程同时想要删除链表的头结点dllFunc2(ls,1)，并在删除结点后释放（delete p）这段空间，当一个线程在定位到first指针之后，就立即被其他一个线程切出，这些线程在删除过程中也一次被切出，而在第一个线程切回并成功释放头指针指向的空间后，其他进程的指针仍指向原来first指针指向的空间，因此无法获得remove后的结点的值，发生程序错误，终止程序。

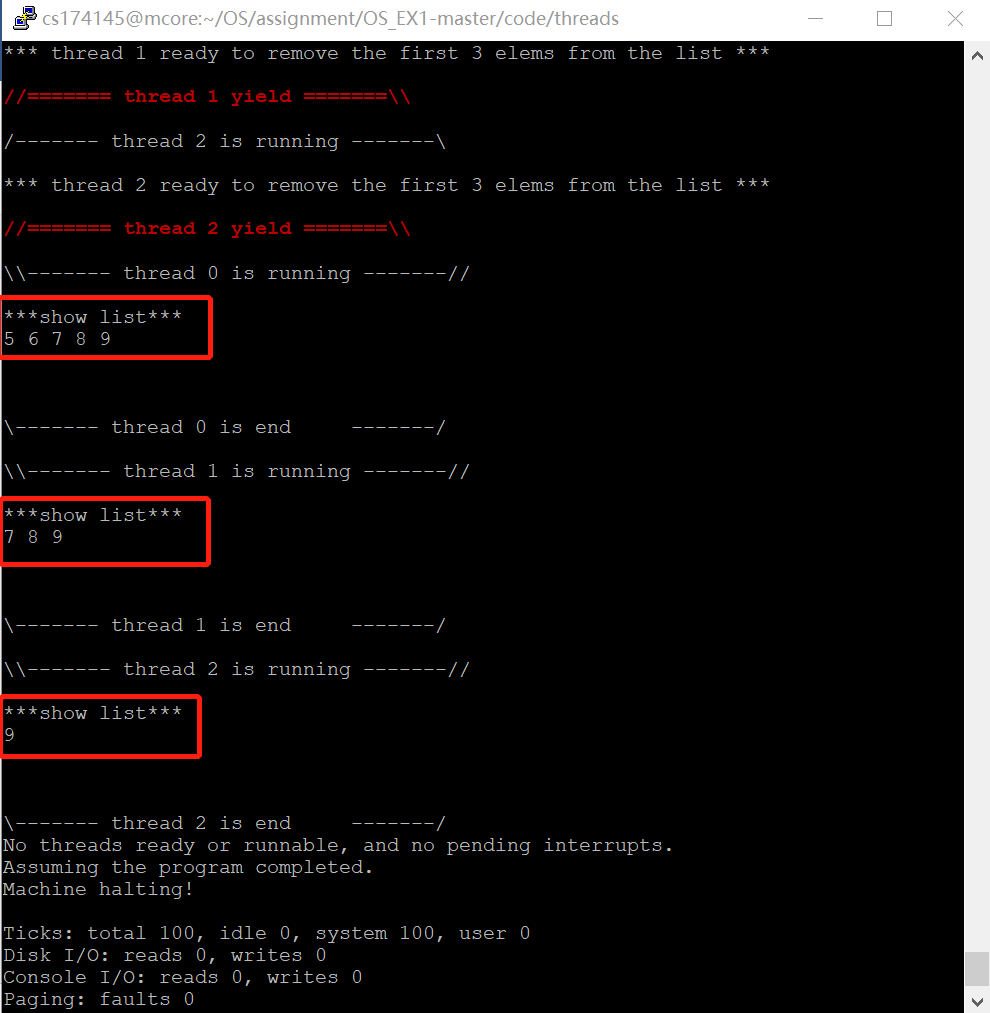
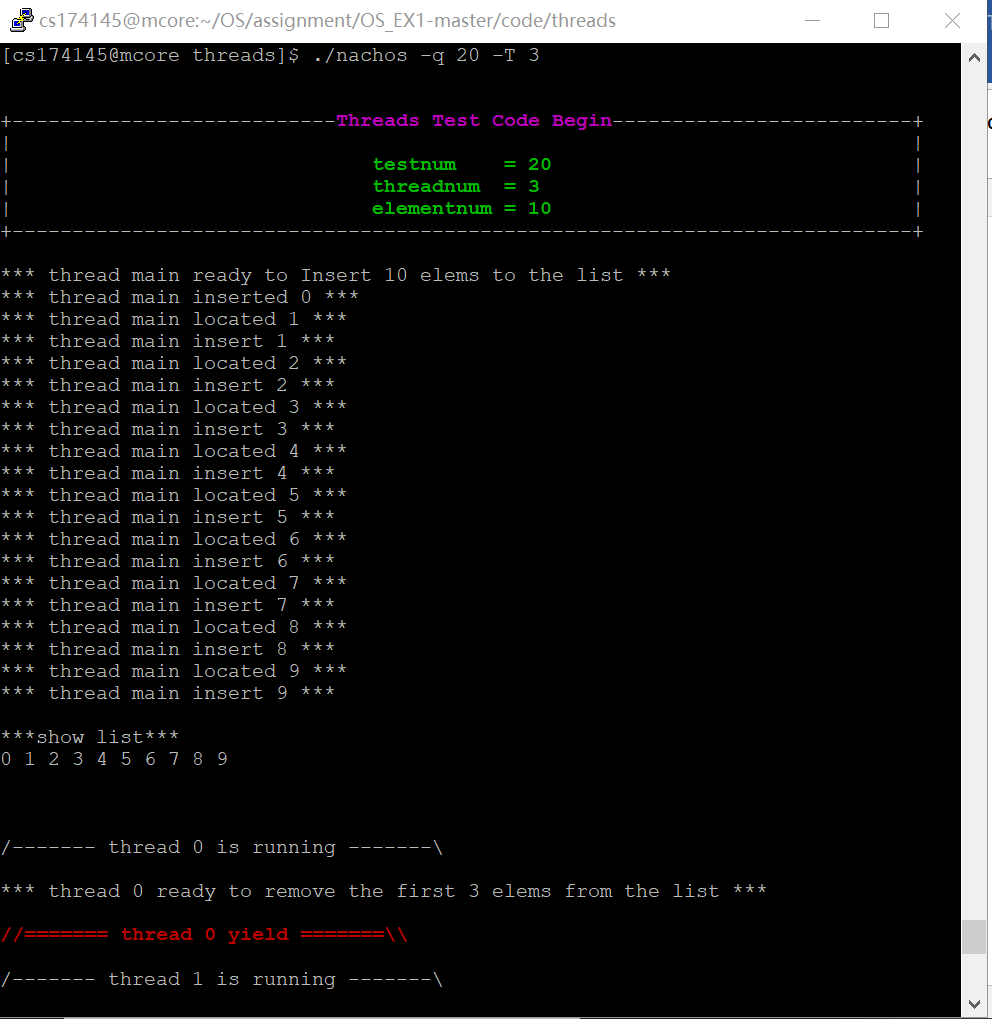


|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| threadmain | thread0 | thread1 | thread2 | list |
|  |  |  |  | NULL |
| dllFunc1 10 |  |  |  | 0 1 2 3 4 5 6 7 8 9 |
|  | dllFunc2 1- yield |  |  | 0 1 2 3 4 5 6 7 8 9 |
|  |  | dllFunc2 1- yield |  | 0 1 2 3 4 5 6 7 8 9 |
|  |  |  | dllFunc2 1-yield | 0 1 2 3 4 5 6 7 8 9 |
|  | dllFunc2 1  -continue |  |  | 1 2 3 4 5 6 7 8 9 |
|  |  | dllFunc2 1  -continue |  | 已放弃 |

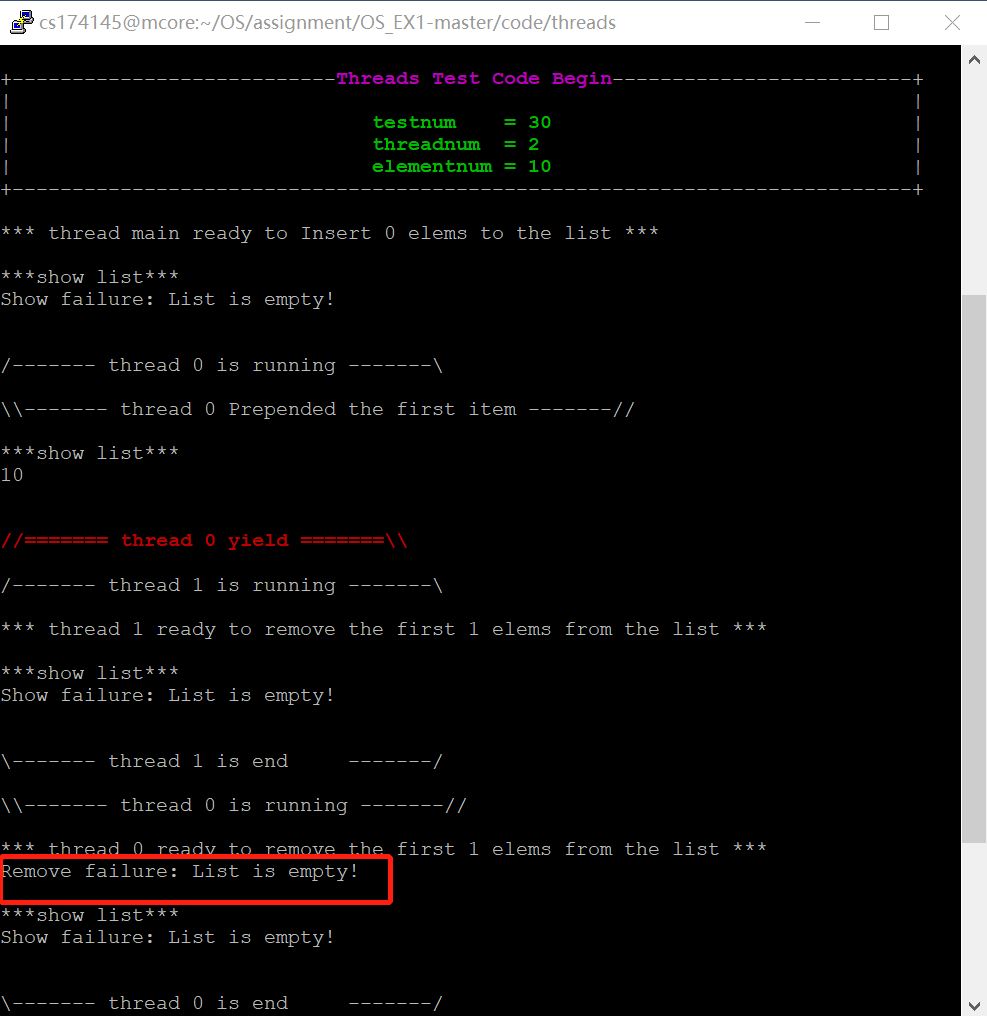
# 实验结果

## 非致命错误

➢Remove乱序./nachos -q 20 -T 3

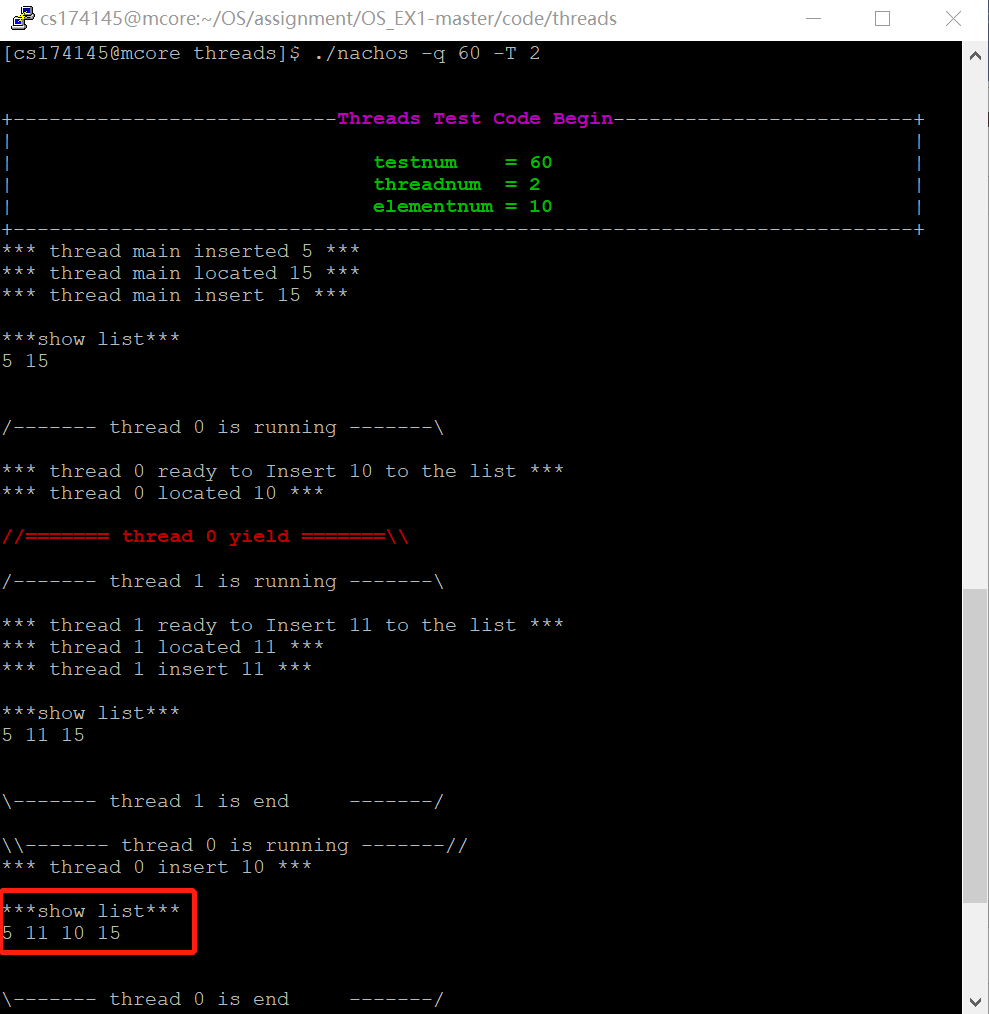


➢Remove失败./nachos -q 30 -T 2



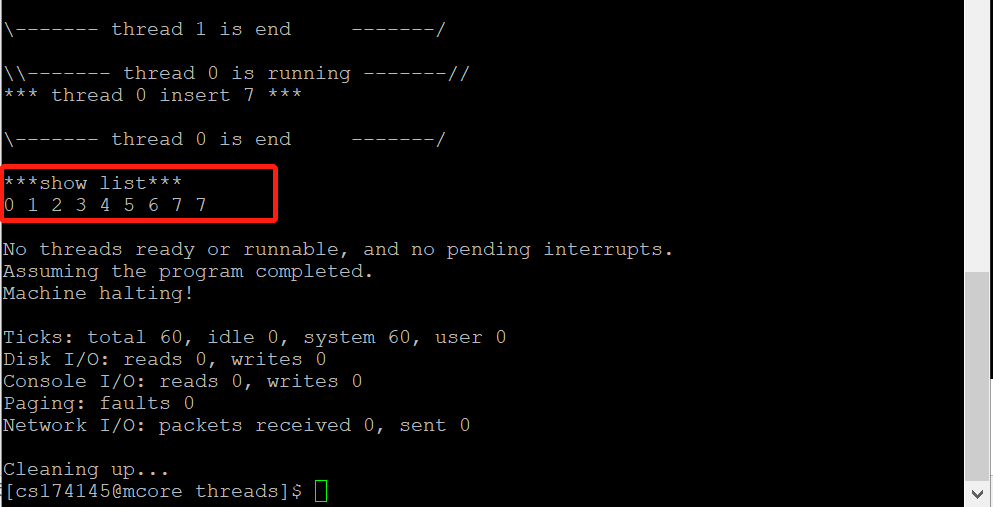
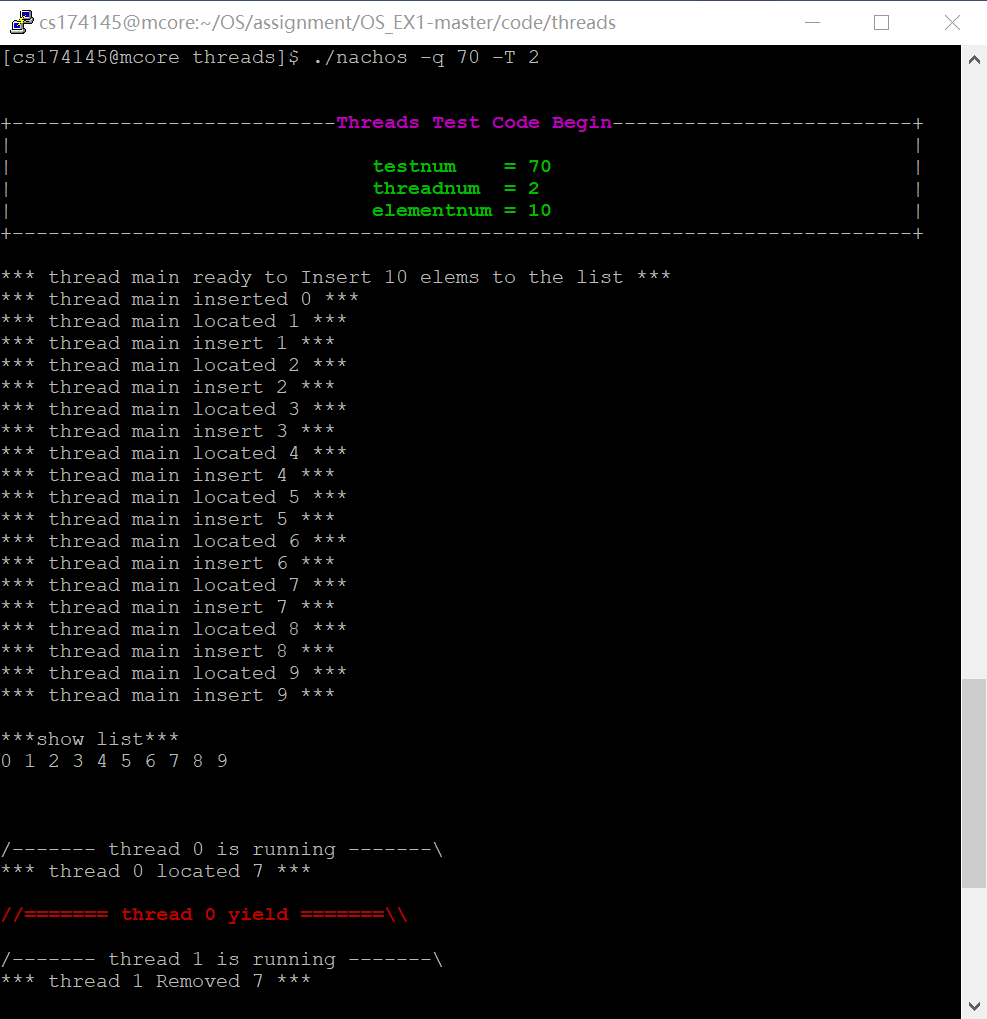
## 元素排序错误

➢乱序错误./nachos -q 60 -T 2

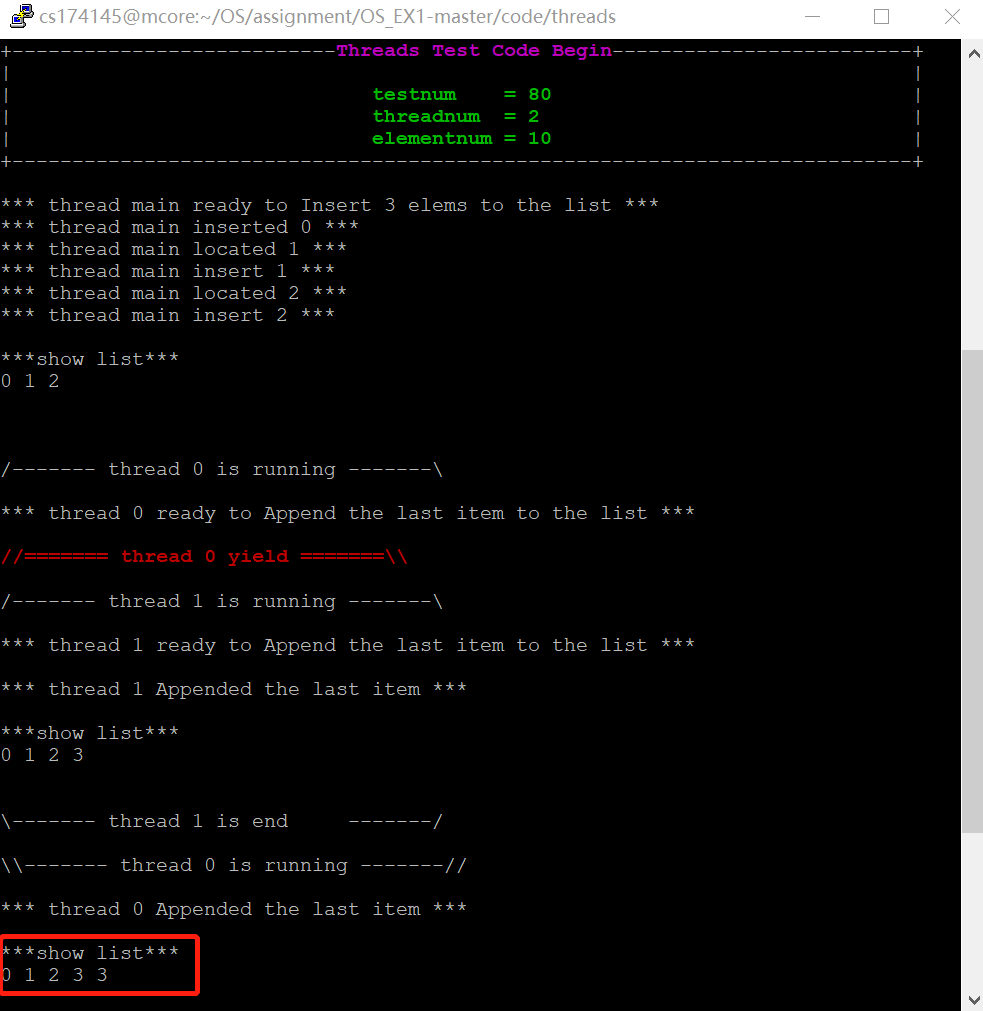


## 插入错误

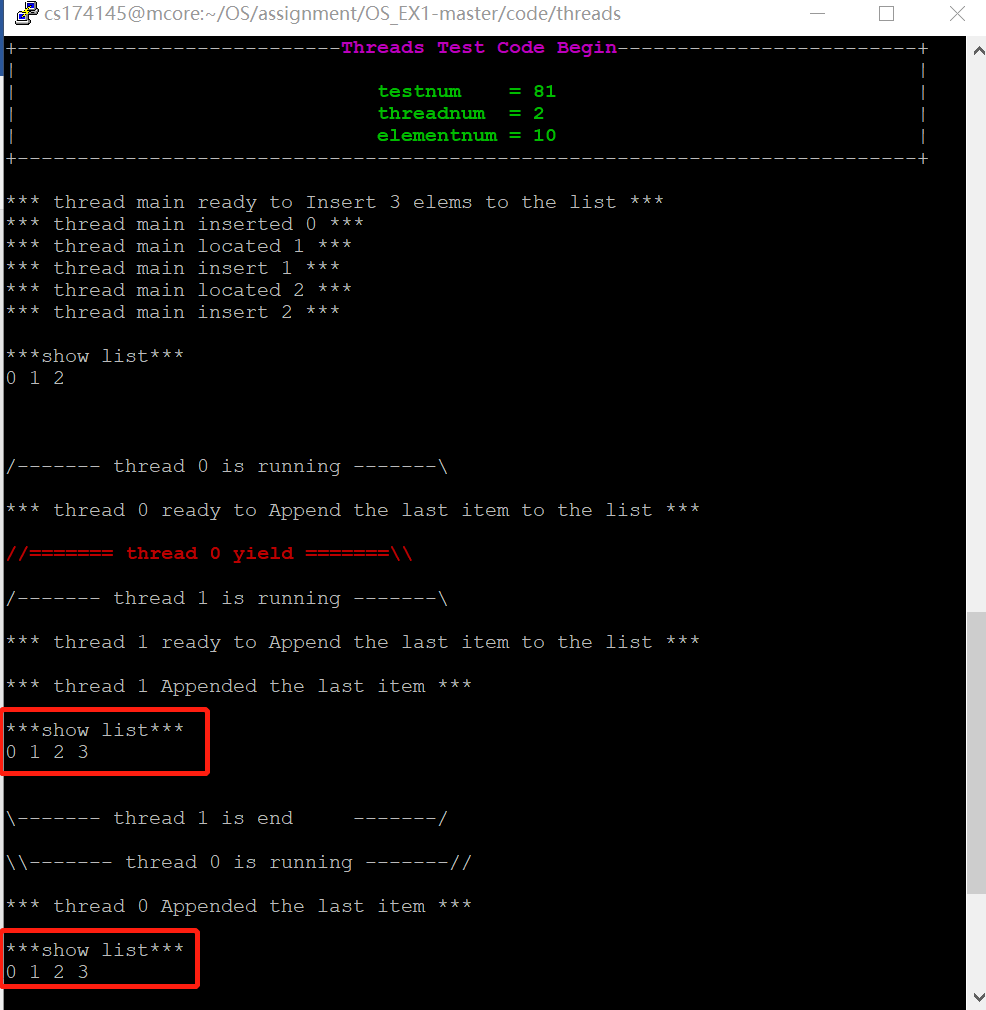
➢链表断裂./nachos -q 70 -T 2



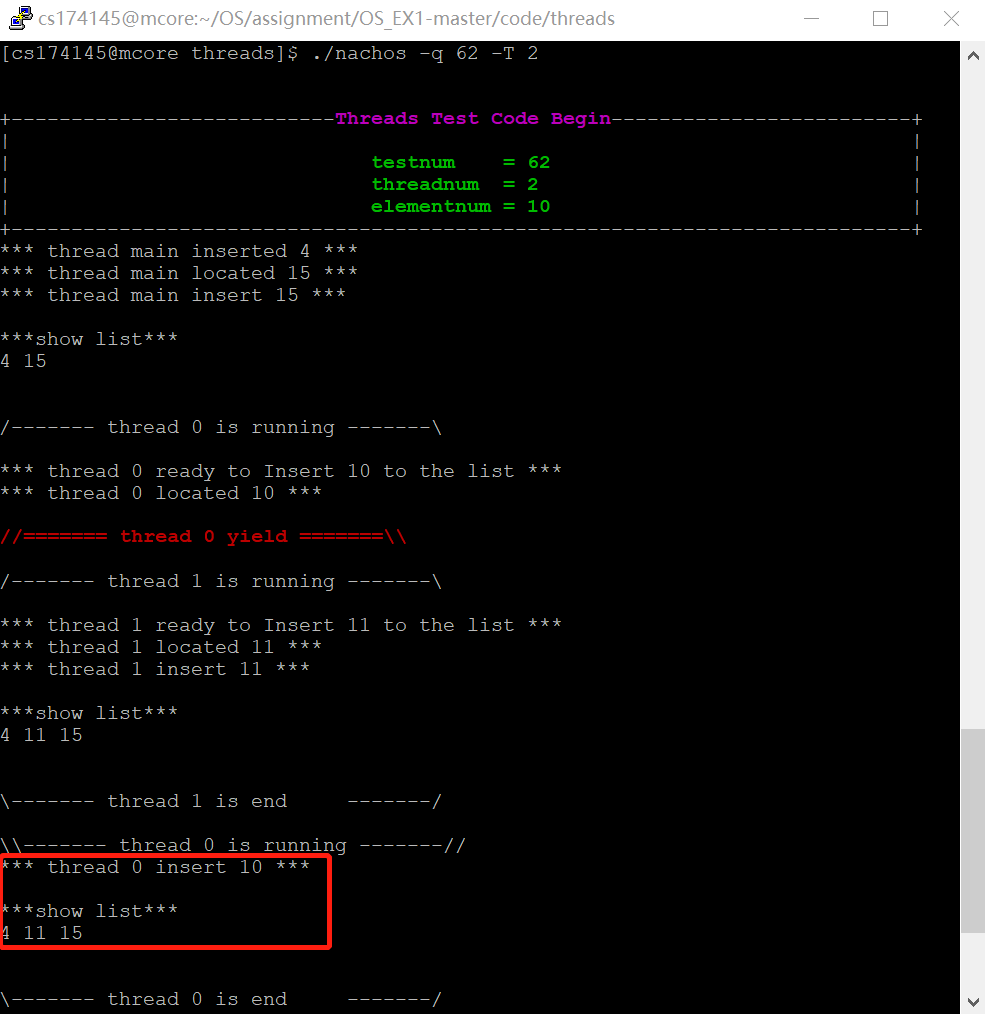
➢插入重复./nachos -q 80 -T 2



➢插入覆盖./nachos -q 81 -T 2 ——链表结构错误

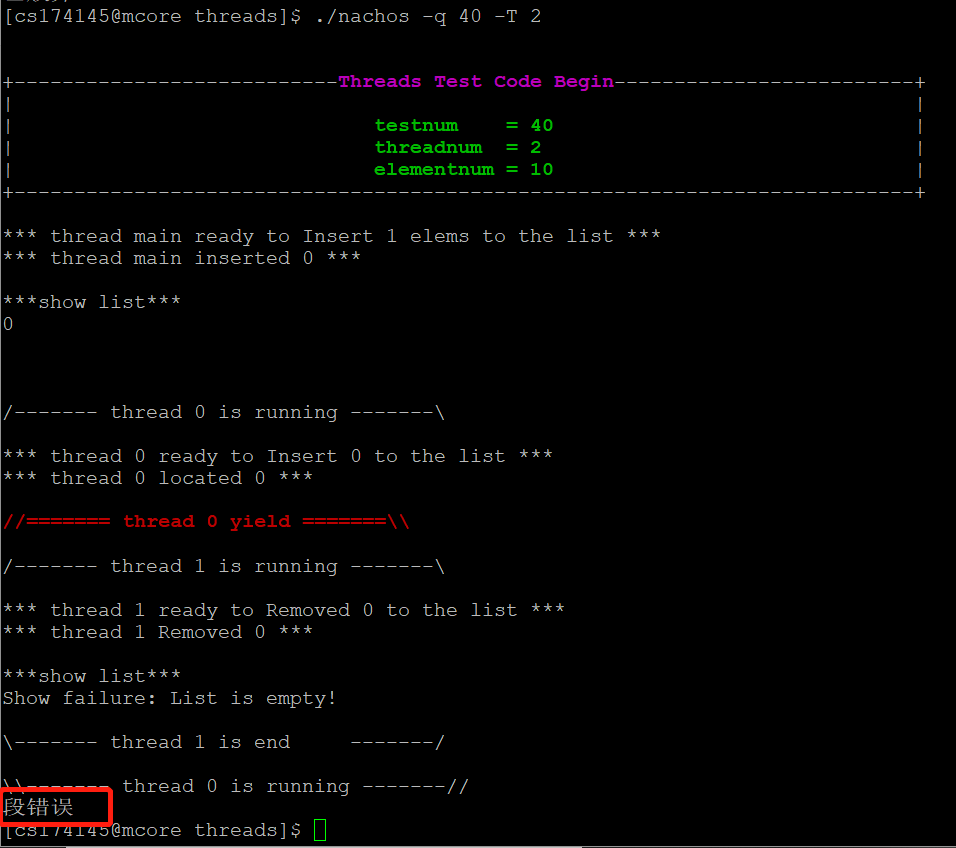


➢插入覆盖./nachos -q 62 -T 2 ——链表结构正常



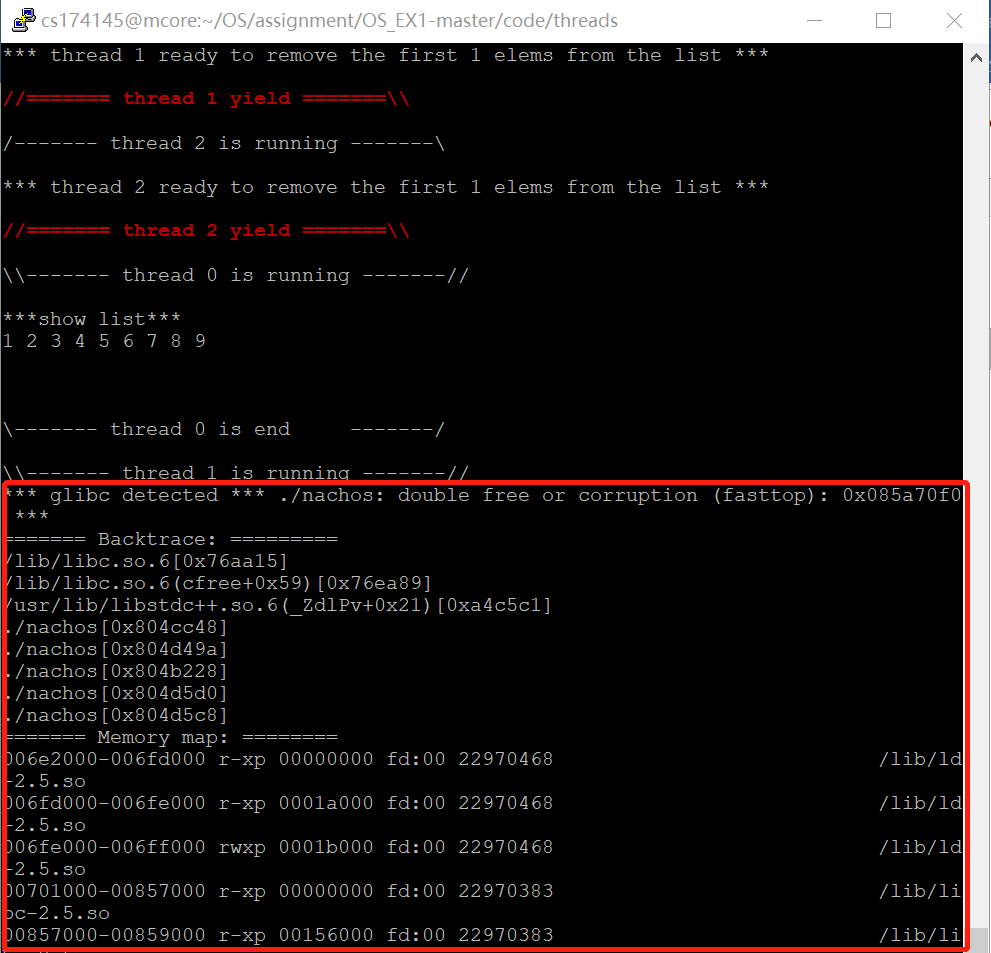
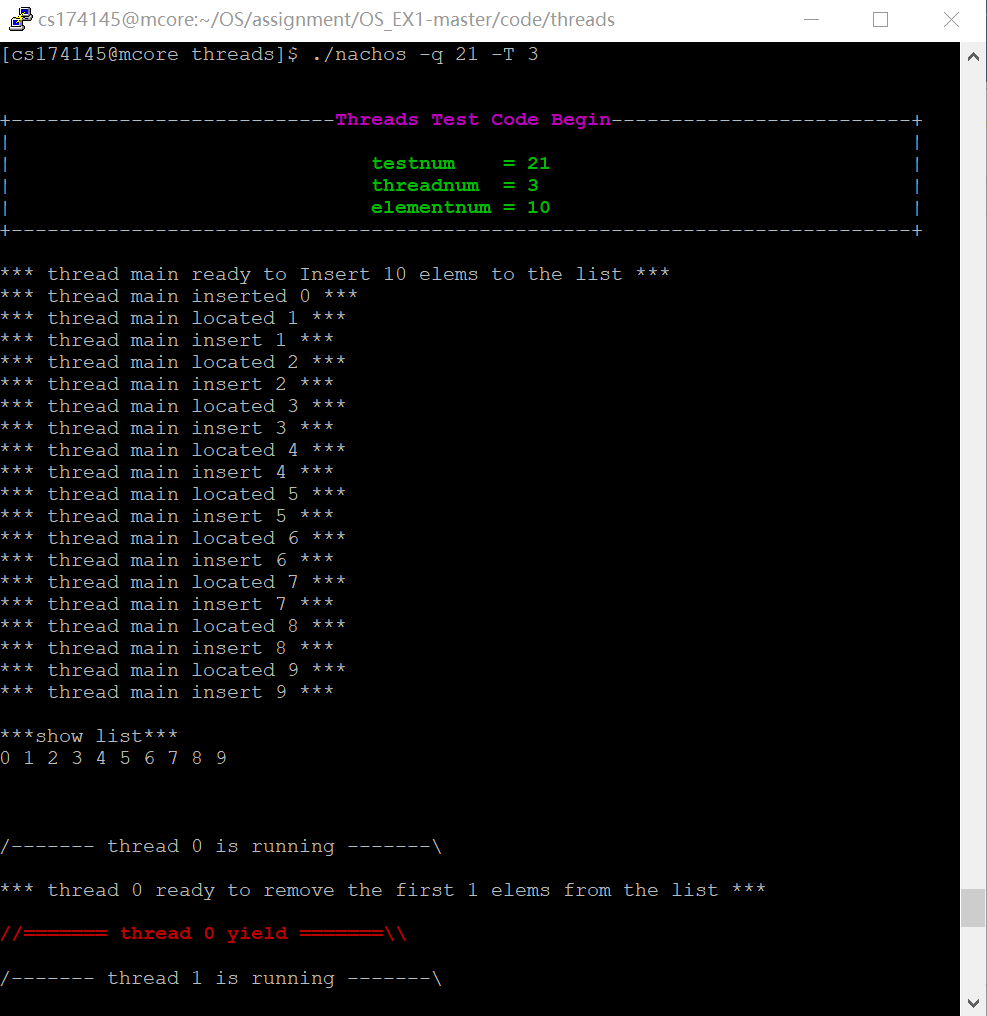
## 段错误

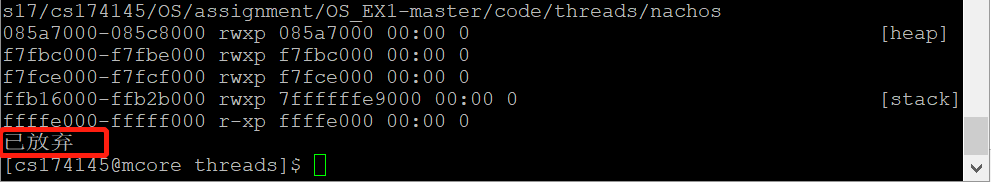
➢./nachos -q 40 -T 2



## 未知错误

➢./nachos -q 21 -T 3





# 实验总结

### 实验分工

在组长的带领和统筹之下，每一个小组成员都参与了全程实验的每一个过程，由于本小组的人员比较少，所以每一个人的工作量都很大。

在实验的过程中，初期的实验安排的阅读材料的理解，三名小组成员进行充分的讨论并达成一致；中期代码编写过程中，我们通过老师推荐的GitHub平台，建立了自己小组的repository，并在上面更新自己的修改的代码，每一个小组成员都贡献了自己的部分，并对其他成员的代码进行理解和修改，在出错测试中，小组成员先集体讨论出错情况，然后分工，每个人都独立编写不同的情况；在后期的报告中，小组成员也是平均分配，共同完成报告的编写，然后汇总和排版。

### 错误及总结

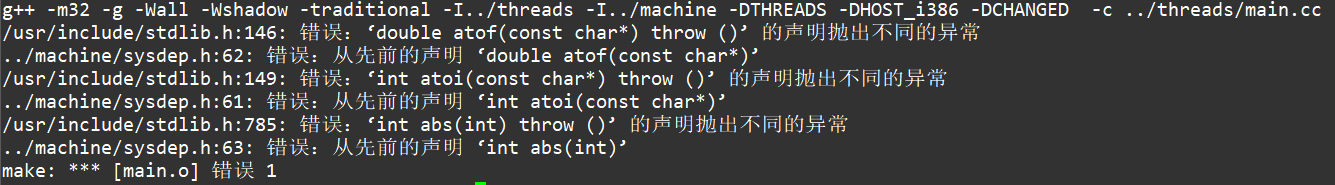
##### 虚拟机环境配置

由于访问实验室的服务器的VPN时常不稳定，因而在本地配置了服务器的虚拟机镜像。最开始的图形化界面镜像由于编译器版本过高等问题，与实验室服务器环境不兼容，因此不能成功编译nachos源码。后来改为使用1.1G的无图形界面的镜像。

为了方便编写代码，需要配置访问端口，使得可以通过物理机上的代码编辑器或Putty访问虚拟机服务器，更真实的模拟实验室上机环境。查看虚拟机IP时可以通过命令hostname -I（大写的i），由于大写的i与大写的L看混了，且hostname -L也能正确执行，所以一直得到的是错误的ip，因为不能正确访问虚拟机。至此深感base58编码的优越性。

##### 在服务器编译报错

虽然虚拟机可以成功编译nachos源码，但是由于虚拟机的编译器版本（4.8.5）仍然高于实验室服务器的编译器版本（4.1.2），所以导致可以在虚拟机中编译的代码到服务器中又不可以成功编译了，比如引用<iostream>库文件就会报以下错误：



所以我们将所有cout输出替换为printf解决了上述问题，编译器版本问题是导致代码修改大规模修改的一个主要原因。

##### 在虚拟机与服务器上线程错误测试结果不同步

出错设计的未知错误中，如case21，虚拟机上的运行结果与服务器上出现不同步，即此错误情况在虚拟机上运行属于非致命错误，但是在服务器上却报错并终止程序。

分析后发现，根源在于当first指针在移动到下一个位置后，其指向的区域为空时，其返回值是否是NULL。在虚拟机环境中，这个为空的值是可以正常在递归调用中返回的，由于first是全局变量，因此其他线程中remove可以继续执行，因此线程的并发可以正常进行；在服务器中，其返回值将直接报错，导致在thread1中，程序直接终止。

##### 讨论用dllist-driver中的两个函数

英文指导文档中提示，每个ThreadTest中调用dllist-driver中两个函数dllFunc1()和dllFun2()，然而在我们实际设计并发错误的过程中，只调用两个dllFunc给并发错误的设计带来的很大的局限，组员设想的并发错误情况无法完全表达出来。

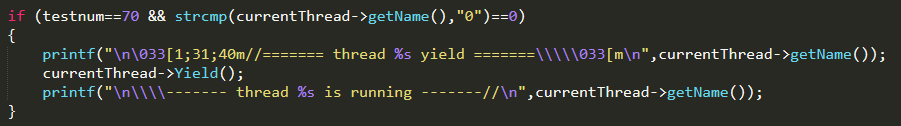
最后经过讨论决定多线程操作同一个链表ls，用dllFunc1()初始化链表数据后，能够自由调用DLList中的方法使各个线程自由对链表操作而不仅仅局限在dllFunc2()中Remove十次。这样使得并发错误的设计更加灵活，能讨论和总结的余地也更大。

##### Thread->Yield()在各个C文件的插入

由于对线程的切入和切除需要考虑到更细化的程度，因此除了在ThreadTest.c中调用Thread->Yield()，还需要进入dllist-driver.c和DLList.c中插入Thread->Yield()来破坏链表插入、删除、判空等操作的原子性。

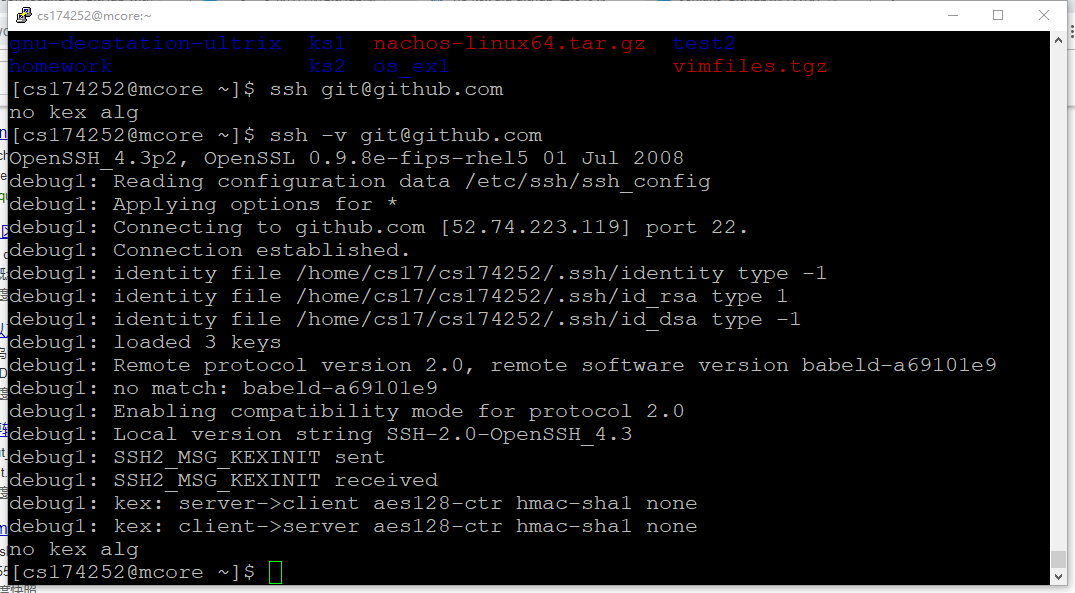
然而这样做又出现了一个问题，如果不加区别地往dllist-driver.c和DLList.c中插入Thread->Yield()，所有调用了两个dllFunc和DLList类的线程都会在同一个位置被切出，导致并发错误设计极不灵活。

小组成员讨论后选择的方法是，将currentThread和testnum作为extern变量在几个C文件中共享，当要往dllist-driver.c和DLList.c中插入Thread->Yield()时先进行currentThread的线程名判断和测试用例编号testnum判断，这样就能使多个测试用例公用一份DLList的文件，并且在调用currentThread->Yied()前后分别输出提示信息，提高设计的可视化并方便调试。



##### 服务器端GitHub和ssh配置

小组成员采用GitHub进行代码协作和版本控制，但是尝试在59.77.8.124服务器端通过ssh连接GitHub远程仓库，配置完整后报错如下：



经与老师沟通后发现是服务器端ssh版本太低问题，并且因为原版本的源已经关闭了无法更新，因为疫情目前又无法进实验室重装，因此目前到服务器端的代码传送还需要通过本地主机从GitHub远程仓库pull下来后用FileZilla或者scp工具传到服务端，较为不方便。建议老师有机会可以重装服务器ssh服务，或者下次实验会尝试在用户目录下自己安装ssh服务。

### 收获心得

本次实验是本学期操作系统原理课程的第一次实验，让我们对nachos的工作过程有了基本地认识，并且了解了线程的工作模式，我们知道了怎么使用yield()函数切换线程和在nachos中适当修改自己的代码；通过实验前对实验材料以及nachos代码的阅读和理解，提高了我们的自学能力。

在完成实验的过程中，小组成员每个过程都进行了非常充分的讨论，有很多时候都会有不同的理解或者疑问，但是最后通过询问老师们得到解决并达成一致，每一个成员都非常了解每次实验的步骤和实验方法。当然在实验过程中，我们也遇到了一些bug，比如一开始我们打算在虚拟机上进行代码编译，但是由于环境问题，没有成功，或者在解压文件的过程中，发现在服务器上运行make clean总会报错，等等。这些问题有些容易解决，有些我们讨论了很长时间，但是正是这些出现的bug，让我们对nachos的运行过程和实验内容有了更加充分的理解，提高了我们的学习能力，也让我们体会到了操作系统实验的乐趣。尽管我们的实验仍存在有一些不足，比如我们对nachos中的debug模块还没有充分的理解，但我们会在日后的学习中，慢慢的体会。

这次实验完成后，我们对线程的并发机制有了更形象的理解，我们知道了设计一个稳定可靠的线程并发是一件很有难度的事情，因为每一个不同的执行顺序都极大可能的造成程序错误，甚至有些是不可恢复的严重错误，如段错误等。本次实验我们小组还想到了更多的可能发生错误的线程切换，最后经过讨论，只留下了比较典型的几种错误。