



# 实 验 报 告

课 程: 操作系统原理

实验标题: 体验 nachos 下的并发程序设计

专 业: 计算机科学与技术

时 间: 2021/3/18 - 2021/4/15

# 目 录

一 实验目的	1
二 实验内容及步骤	1
2.1 安装 nachos, 阅读源代码	1
2.2 用 C++ 实现双向有序链表	2
2.2.1 概述	2
2.2.2 Makefile 修改	2
2.2.3 代码实现及说明	3
dlist.cc	3
dlist-driver.cc	6
修改 nachos 的 threadtest.cc 和 main.cc	7
2.2.4 实现效果	8
2.3 体验 nachos 线程系统	9
2.3.1 实验任务	9
2.3.2 实验设计	10
期望结果	10
执行顺序设计	10
2.3.3 代码实现、说明及效果	11
Case 1: 同时向表头插入一个最小的元素, 导致其中一个元素未被成功插入	11
Case 2: 在同一位置插入元素, 导致链表 Key 有序性被破坏	12
Case 3: 同时向表头插入元素和删除元素, 导致删除了非表头元素	14
Case 4: 同时删除表头元素, 由于节点已被析构导致段错误	14
三 实验总结与感想	15

# 实验一 • 体验 nachos 下的并发程序设计

实验时间 2021/3/18 - 2021/4/15

## 一 实验目的

本次实验的目的在于对 nachos 进行熟悉，并初步体验 nachos 下的并发程序设计。实验内容分三部分：

- 安装 nachos，并阅读源代码
- 用 C++ 实现双向有序链表
- 体验 nachos 线程系统，在 nachos 系统中使用所写的链表程序并演示一些并发错误

## 二 实验内容及步骤

### 2.1 安装 nachos，阅读源代码

Q. 我需要修改/新增 nachos 哪一部分的代码？我不该随意改动哪些代码？

- 可以修改 `#define` 确定机器或者系统参数的指令（比如，物理内存或者默认堆栈的大小）；
- 可以修改一些测试函数的代码，如本实验中 `ThreadTest` 函数等。item 不该修改实现机器本身的任何代码。

Q. code 文件夹下的子目录都放了啥？输入“make depend”、“make”、和“./nachos”都发生了什么？

- code 文件夹里面存放了 bin、machine、network 等等文件夹以及 Makefile.common、Makefile、Makefile.dep 文件；
- 当输入 make depend 的时候，编译器会自动根据 Makefile 文件里面的内容形成各个头文件以及 c 文件的依赖关系；
- 当输入 make 的时候，编译器对已经形成依赖关系的文件进行编译，生成可执行文件 nachos；
- 当输入 ./nachos 的时候，可执行文件运行。

Q. 我如何 Debug nachos 代码？

- 在代码中添加 printf（或者 fprintf）
- 使用 gdb 调试器或者其他调试器
- 插入对 Nachos 提供的 DEBUG 函数调用

Q. 如何在 nachos 进行 context switch？

- 在代码中使用 `currentThread->Yield()`；完成强行线程切换
- 使用具有 -rs 标志的非自愿上下文切换

Q. nachos 中线程的概念（用户级线程还是内核级线程？）及其运行方式？

用户级线程，是指用户程序自行采用线程管理库，在应用本身实现线程管理功能；这样的线程是操作系统不可感知的；而内核级线程是由操作系统来管理的。在 nachos 中的 Thread 内实现了线程相关的管理逻辑，并不需要由用户程序自己实现线程管理，因此 nachos 的线程是内核级线程。

## 2.2 用 C++ 实现双向有序链表

### 2.2.1 概述

本实验需要自行编写三个文件 `dllist.h`，`dllist.cc` 和 `dllist-driver.cc`。这三个文件的作用说明如下：

- `dllist.h`：由 nachos 的英文文档给出，定义了 `DLL_Element` 类（表示链表节点）和 `DLLList` 类（表示链表本身）；本实验需要实现这两个类中定义的方法。
- `dllist.cc`：包含 `dllist.h` 中定义的 `DLL_Element` 和 `DLLList` 类各方法的实现代码。
- `dllist-driver.cc`：包含了两个工具函数
  - `PutItem(list, N, threadNumber)`：向双向链表中插入随机的  $N$  个元素
  - `RemoveItem(list, N, threadNumber)`：连续从双向链表首部删除  $N$  个元素

### 2.2.2 Makefile 修改

nachos 使用 Makefile 来管理编译依赖、编译链接规则等。由于本实验中向 nachos 项目中添加了新的三个源码文件，为了让它们能被正常编译链接、并与 nachos 结合，这里需要修改关于 threads 子目录的编译规则，该规则位于 nachos 项目根目录的 `Makefile.common` 中（这个文件定义了需要使用 gcc 编译的文件，和每个可执行文件需要链接的链接文件）。具体地，需要修改此文件的这些内容：

- 在 `THREAD_C` 中，添加源代码文件 `dllist.cc` 和 `dllist-driver.cc`：

```
54 THREAD_C = ../threads/main.cc\  
55     ../threads/list.cc\  
56     ../threads/scheduler.cc\  
57     ../threads/synch.cc \  
58     ../threads/synchlist.cc\  
59     ../threads/system.cc\  
60     ../threads/thread.cc\  
61     ../threads/utility.cc\  
62     ../threads/threadtest.cc\  
63     ../machine/interrupt.cc\  
64     ../machine/sysdep.cc\  
65     ../machine/stats.cc\  
66     ../machine/timer.cc\  
67     ../threads/dllist.cc\  
68     ../threads/dllist-driver.cc
```

- 在 `THREAD_H` 中，添加头文件 `dllist.h`：

```

40 THREAD_H =../threads/copyright.h\
41  ../threads/list.h\
42  ../threads/scheduler.h\
43  ../threads/synch.h \
44  ../threads/synclist.h\
45  ../threads/system.h\
46  ../threads/thread.h\
47  ../threads/utility.h\
48  ../machine/interrupt.h\
49  ../machine/sysdep.h\
50  ../machine/stats.h\
51  ../machine/timer.h\
52  ../threads/dllist.h

```

- 在 THREAD\_O 中，添加由 dllist.cc 和 dllist-driver.cc 编译后产生的链接文件 dllist.o 和 dllist-driver.o:

```

72 THREAD_O =main.o list.o scheduler.o synch.o synclist.o system.o thread.o \
73  utility.o threadtest.o interrupt.o stats.o sysdep.o timer.o dllist.o dllist-driver.o

```

完成后，输入 `make depend && make` 指令，检查是否正常重新生成可执行文件。

### 2.2.3 代码实现及说明

#### dllist.cc

根据题目的要求，我们需要使用 C++ 实现一个双向链表，而 dllist.h 中已经给出了类定义和方法定义。具体地，我们需要在 dllist.cc 中实现的方法、这些方法的作用及实现思路阐述如下：

- `DLLElement(void *itemPtr, int sortKey)` : DLLElement 类的构造函数  
**实现思路：**简单将 item, key 设置为参数给定的值，然后设置指向 prev, next 为 NULL 即可。
- `DLList::DLList(), DLList::~~DLList()` : DLList 类的构造函数和析构函数

**实现思路：**

- 在构造函数中，初始化 first, last 指针为 NULL;
- 在析构函数中，将链表中的所有元素 DLLElement 析构，然后将 first, last 设置为 NULL。

- `void DLList::Prepend(void *item)` : 在双向链表的首部插入值为 item 的元素，并自动为新元素分配一个小于链表首部原有元素的 key 值的 key 值。

**实现思路：**

- 如果双向链表为空即 `first == NULL`，则创建一个 key 值为 0 的元素，并将指针赋给 first, last。
- 否则，创建元素；为了保证 key 的有序性，将新元素的 key 设为 `first->key - 1`，然后修改新元素的 next 指向和 first 指针的 prev 指向，最后更新 first 指针。

- `void DLList::Append(void *item)` : 在双向链表的尾部插入值为 item 的元素，并自动为新元素分配一个大于链表尾部原有元素的 key 值的 key 值。

**实现思路：**同上，将上面插入到首部的步骤对偶即可。

- `void* DLList::Remove(int *keyPtr)` : 从双向链表的头部删除一个元素，并将被删除元素的 key 值传给参数指针。

**实现思路：**将 first 指向 `first->next`，然后将原先的 first 指向的元素析构即可。

- `bool DLLList::IsEmpty()` : 当双向链表不为空时, 返回 true  
实现思路: 当 `first == last == NULL` 时, 表示双向链表为空。
- `void DLLList::SortedInsert(void *item, int sortKey)` : 将 key 值为 sortKey 的元素插入到双向链表的正确位置中。

实现思路:

- 如果链表为空, 直接插入到首部并修改 first, last 的指向
- 否则, 从 first 开始遍历链表, 直到找到 `x->key > key` 的元素 x 或找不到这样的元素为止。
- 如果找不到这样的元素, 即链表中所有元素的 key 都比给定的 key 少, 则将其插入到表尾, 修改 `last->next` 和 `newElement->prev`, 然后修改 last 的指向。
- 如果找到了元素 x, 则更新 `newElement->prev`, `newElement->next`, `x->next->prev` 和 `x->next`。

- `void* DLLList::SortedRemove(int sortKey)` : 删除 key 值为 sortKey 的元素 (如果存在)。

实现思路: 从首部遍历, 找到符合条件的节点, 修改该节点前、后节点的连接关系即可。

具体代码实现如下:

```

1  #include "stdio.h"
2  #include "stdlib.h"
3  #include "assert.h"
4  #include "dllist.h"
5  #include "thread.h"
6
7  extern Thread *currentThread;
8
9  // initialize a list element (constructor, destructor)
10 DLLElement::DLLElement(void *itemPtr, int sortKey)
11 {
12     item = itemPtr;
13     key = sortKey;
14     next = NULL;
15     prev = NULL;
16 }
17
18 DLLList::DLLList()
19 {
20     first = NULL;
21     last = NULL;
22 }
23
24 // in destructor function, we delete all DLLElement(s) in the current list
25 // and set first/last pointer to NULL
26 DLLList::~~DLLList()
27 {
28     DLLElement *ptr;
29     while (first != NULL) {
30         ptr = first;

```

```

31     first = first->next;
32     delete ptr;
33 }
34 last = NULL;
35 }
36
37 // add to head of list, set key = min_key - 1
38 void DLList::Prepend(void *item)
39 {
40     DLLElement *curItem = new DLLElement(item, (first == NULL) ? 0 : (first->key
41     ↪ - 1));
42     curItem->next = first;
43     if (first != NULL)
44         first->prev = curItem;
45     first = curItem;
46     last = (last == NULL) ? curItem : last;
47 }
48
49 // add to tail of list, set key = max_key + 1
50 void DLList::Append(void *item)
51 {
52     DLLElement *curItem = new DLLElement(item, (last == NULL) ? 0 : (last->key +
53     ↪ 1));
54     curItem->prev = last;
55     if (last != NULL)
56         last->next = curItem;
57     last = curItem;
58     first = (first == NULL) ? curItem : first;
59 }
60
61 // remove from head of list
62 // set *keyPtr to key of the removed item
63 // return item (or NULL if list is empty)
64 void* DLList::Remove(int *keyPtr)
65 {
66     if (first == NULL)
67         return NULL;
68     *keyPtr = first->key;
69     DLLElement *removedItem = first;
70     last = (first == last) ? NULL : last;
71     first = first->next;
72     if (first != NULL)
73         first->prev = NULL;
74     return removedItem;
75 }
76
77 // return true if list has elements
78 bool DLList::IsEmpty()
79 {
80     return (first != NULL && last != NULL);
81 }

```

```

80
81 // routines to put/get items on/off list in order (sorted by key)
82 void DLList::SortedInsert(void *item, int sortKey)
83 {
84     if (first == NULL || sortKey <= first->key) {
85         Prepend(item);
86         first->key = sortKey;
87         return;
88     }
89
90     DLLElement *newItem = new DLLElement(item, sortKey),
91         *ptr = first;
92     while (ptr != NULL && ptr->key < sortKey)
93         ptr = ptr->next;
94     if (ptr == NULL) {
95         last->next = newItem,
96         newItem->prev = last,
97         last = newItem;
98     }
99     else {
100         ptr->prev->next = newItem,
101         newItem->prev = ptr->prev,
102         newItem->next = ptr,
103         ptr->prev = newItem;
104     }
105 }
106
107 void* DLList::SortedRemove(int sortKey)
108 {
109     DLLElement *ptr = first;
110     assert(first != NULL);
111     while (ptr != NULL && ptr->key < sortKey)
112         ptr = ptr->next;
113     if (ptr == NULL || ptr->key != sortKey)
114         return NULL;
115     if (ptr->next != NULL)
116         ptr->next->prev = ptr->prev;
117     if (ptr->prev != NULL)
118         ptr->prev->next = ptr->next;
119     return ptr->item;
120 }

```

### dllist-driver.cc

根据题目需求，我们还需要实现两个工具函数，其中一个函数生成 N 个随机 key 的元素并插入双向链表；另一个函数从链表首部依次移除 N 个元素。具体代码实现如下：

```

1 void PutItem(DLList *list, int n, int threadNumber)
2 {
3     if (!isSranded) {
4         isSranded = true;
5         srand(time(NULL));

```



```

6     }
7     for (int i = 0; i < n; i++) {
8         int key = rand() % 1000;
9         printf("[Thread %d] insert: %d\n", threadNumber, key);
10        list->SortedInsert((void*)key, key);
11        list->foreach(0);
12    }
13 }
14
15 void RemoveItem(DLLList *list, int n, int threadNumber)
16 {
17     for (int i = 0, key; i < n; i++) {
18         printf("[Thread %d] remove: ", threadNumber);
19         list->Remove(&key);
20         printf("%d\n", key);
21         list->foreach(0);
22     }
23 }

```

### 修改 nachos 的 threadtest.cc 和 main.cc

为了让我们编写的代码能够通过 nachos 运行，需要修改 *main.cc* 和 *threadtest.cc*，添加相关的函数调用和运行参数，以便于测试。首先，对 *threadtest.cc* 做出如下修改：

- 定义全局变量 `numberCount` 表示每轮次随机生成的数字个数，默认 `numberCount = 10`。
- 添加两个函数 `ThreadTest2()` 和 `DLLListThread()` 作为测试的入口。

为了下一部分实验的方便，在这里同时做了以下事情：

- 定义全局变量 `threadNum` 表示同时运行的线程数。
- 在 `ThreadTest2()` 中实现线程的创建和运行。

```

1 void DLLListThread(int threadNum)
2 {
3     PutItem(list, numberCount, threadNum);
4     RemoveItem(list, numberCount, threadNum);
5 }
6
7
8 void ThreadTest2()
9 {
10    DEBUG('t', "Entering ThreadTest2");
11
12    DEBUG('t', "Creating DLLList()");
13    list = new DLLList();
14
15    DEBUG('t', "Creating fork thread");
16    for (int i = 1; i < threadCount; i++) {
17        Thread *t = new Thread("fork thread");
18        t->Fork(DLLListThread, i);
19    }
20

```

```

21     DLListThread(0);
22
23     if (list != NULL)
24         delete list;
25 }

```

然后在 *main.cc* 中，添加自定义参数。nachos 已经预定义了一部分参数，如 -q 用于指定测试的编号。这里再添加 -n 参数用于指定 `numberCount`：

```

1  for (argc--, argv++; argc > 0; argc -= argCount, argv += argCount) {
2      argCount = 1;
3
4      switch (argv[0][1]) {
5          case 'q':
6              testnum = atoi(argv[1]);
7              argCount++;
8              break;
9          case 'n':
10             numberCount = atoi(argv[1]);
11             argCount++;
12             break;
13          case 't':
14             threadCount = atoi(argv[1]);
15             argCount++;
16          default:
17             break;
18      }
19 }

```

## 2.2.4 实现效果

添加相应的 `verbose` 函数显示链表的插入和删除过程：

```

1  void DLList::foreach(bool showRelation)
2  {
3      int sz = 0, flag = 0;
4      DLLElement *ptr = first;
5      printf("\t ");
6
7      while (ptr != NULL) {
8          flag = 1;
9          if (sz != 0)
10             printf("->");
11             printf("%d", ptr->key);
12             if (showRelation) {
13                 printf("(");
14                 if (ptr->prev != NULL)
15                     printf("p=%d", ptr->prev->key);
16                 if (ptr->prev != NULL && ptr->next != NULL)

```

```

17         printf(" ");
18         if (ptr->next != NULL)
19             printf("n=%d", ptr->next->key);
20         printf(")");
21     }
22     sz++, ptr = ptr->next;
23 }
24 if (flag)
25     putchar(' ');
26 printf("(total: %d)\n", sz);
27 }

```

然后执行 make 命令编译新的 nachos 可执行文件，执行命令：

```

1 ./nachos -q 2 -t 1 -n 5 # 运行 ThreadTest2, 单线程运行, 每轮次生成 5 个数字

```

效果如下：

```

# cs182204269 @ mcore in threads [19:54:54]
$ ./nachos -q 2 -n 5 -t 1
[Thread 0] insert: 710
710 (total: 1)
[Thread 0] insert: 435
435->710 (total: 2)
[Thread 0] insert: 544
435->544->710 (total: 3)
[Thread 0] insert: 69
69->435->544->710 (total: 4)
[Thread 0] insert: 673
69->435->544->673->710 (total: 5)
[Thread 0] remove: 69
435->544->673->710 (total: 4)
[Thread 0] remove: 435
544->673->710 (total: 3)
[Thread 0] remove: 544
673->710 (total: 2)
[Thread 0] remove: 673
710 (total: 1)
[Thread 0] remove: 710
(total: 0)
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 140, idle 0, system 140, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...

```

## 2.3 体验 nachos 线程系统

### 2.3.1 实验任务

修改 Nachos 源代码体验并发程序问题：

- 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。

- 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；
- 实现以上执行顺序：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，重新编译 threads 子系统并运行观察出现的问题。

## 2.3.2 实验设计

### 期望结果

在此任务中，我们假定程序拥有两个线程 Thread 0 和 Thread 1，分别执行链表的顺序插入和首部删除操作各 5 次。

为了下文说明的便利，我们期望两个线程正确的执行顺序如下：

- Thread 0 和 Thread 1 默认交替执行插入和删除操作。
- 任一线程执行插入操作 `PutItem` 时，总是随机生成一个数插入到链表中的对应位置，分配空间并维护节点链接；插入操作完成后，才切换至另一线程；
- 任一线程执行删除操作 `RemoveItem` 时，总是删除链表首部（最小）的元素，维护节点链接和表头指向，然后将被删除的节点析构 (`delete`)；删除操作完成后，才切换至另一线程。

因此，我们期望的运行结果应如下图所示：

```
[cs182204269@mc core threads]$ ./nachos -q 2 -n 5
[Thread 0] insert: 232
232 (total: 1)
[Thread 1] insert: 604
232->604 (total: 2)
[Thread 0] insert: 39
39->232->604 (total: 3)
[Thread 1] insert: 751
39->232->604->751 (total: 4)
[Thread 0] insert: 664
39->232->604->664->751 (total: 5)
[Thread 1] insert: 800
39->232->604->664->751->800 (total: 6)
[Thread 0] insert: 205
39->205->232->604->664->751->800 (total: 7)
[Thread 1] insert: 166
39->166->205->232->604->664->751->800 (total: 8)
[Thread 0] insert: 970
39->166->205->232->604->664->751->800->970 (total: 9)
[Thread 1] insert: 690
39->166->205->232->604->664->690->751->800->970 (total: 10)
[Thread 0] remove: 39
166->205->232->604->664->690->751->800->970 (total: 9)
[Thread 1] remove: 166
205->232->604->664->690->751->800->970 (total: 8)
[Thread 0] remove: 205
232->604->664->690->751->800->970 (total: 7)
[Thread 1] remove: 232
604->664->690->751->800->970 (total: 6)
[Thread 0] remove: 604
664->690->751->800->970 (total: 5)
[Thread 1] remove: 664
690->751->800->970 (total: 4)
[Thread 0] remove: 690
751->800->970 (total: 3)
[Thread 1] remove: 751
800->970 (total: 2)
[Thread 0] remove: 800
970 (total: 1)
[Thread 1] remove: 970
(total: 0)
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
```

### 执行顺序设计

上文所提到的正确执行顺序，都是建立在“插入操作和删除操作都是原子操作”的前提之下的；然而，在实际的情况中并不是这样。具体地说，插入操作可以被分为以下几个阶段：

- 创建 DLLElement 对象 (new)
- 寻找插入位置，并设置新节点的相邻指向
- 修改前、后节点的相邻指向
- 维护表头/表尾

而表头删除操作可以被分为以下几个阶段：

- 修改被删除节点的相邻节点的 prev 指向
- 维护表头/表尾
- 析构被删除的节点 (delete)

如果在上述每个阶段的之间，执行线程的切换，则这样的“错误执行顺序”可能就会引发错误的结果，例如：

- Thread 0 在插入节点、修改相邻指向的阶段时，Thread 1 在当前节点的附近位置插入一个新节点。
- Thread 0 插入了一个最小的元素、在修改表头的指向的阶段时，Thread 1 从表头删除了元素。
- Thread 0 和 Thread 1 同时从表头删除了一个元素。

### 2.3.3 代码实现、说明及效果

**Case 1: 线程 0 和线程 1 同时向表头插入一个最小的元素，导致其中一个元素未被成功插入**  
此场景模拟如下：

- 表头 first 指向节点 node0 对应的 key 为 a
- 线程 0 创建了一个 DLLElement 对象 node1，设置其 key 为 b ( $b < a$ )
- 线程 0 将 first->prev 赋值为 node1
- 发生线程切换
- 线程 1 创建了一个 DLLElement 对象 node2，设置其 key 为 c ( $c < b$ )
- 此时 first 指针尚未被赋值为 node1，线程 1 将 first->prev 赋值为 node2
- 发生线程切换
- 线程 0 将 first 赋值为 node1
- 发生线程切换
- 线程 1 将 first 赋值为 node2
- 发生线程切换

期望结果：node2(c) -> node1(b) -> node0(a), first = node2

实际结果：node2(c) -> node0(a), first = node2

结果对比：node1 未能成功插入链表

为了实现这种执行顺序，在每次向首部插入元素、修改 first 指针之前，调用 `currentThread->Yield()`：

```

1 // add to head of list, set key = min_key - 1
2 void DList::Prepend(void *item)
3 {
4     DLLElement *curItem = new DLLElement(item, (first == NULL) ? 0 : (first->key
5         ↪ - 1));
6     curItem->next = first;
7     if (first != NULL)
8         first->prev = curItem;
9
10    // 执行强制线程切换
11    if (threadTestcase == 1)
12        currentThread->Yield();
13
14    first = curItem;
15    last = (last == NULL) ? curItem : last;
16 }

```

运行结果如图所示。当两个线程同时往链表首部插入元素时，错误的执行顺序会导致其中一个插入的数丢失；而如果在不同的位置插入，则不会发生此问题。

```

[Thread 0] insert: 412
[Thread 1] insert: 918
          412 (total: 1)
          918 (total: 1)
[Thread 0] insert: 754
[Thread 1] insert: 366
          754->918 (total: 2)
          366->918 (total: 2)
[Thread 0] insert: 326
[Thread 1] insert: 425
          366->425->918 (total: 3)
          326->366->425->918 (total: 4)

```

**Case 2: 线程 0 和线程 1 在同一位置插入元素，导致链表 Key 有序性被破坏**  
此场景模拟如下：

- 初始化链表，插入 key 为 1 的节点 node1、key 为 4 的节点 node4
- 线程 0 创建了一个 DLLElement 对象 node3，设置其 key 为 3
- 线程 0 找到节点插入位置为 node4 之前
- 发生线程切换
- 线程 1 创建了一个 DLLElement 对象 node2，设置其 key 为 2
- 由于尚未修改关联，线程 1 找到节点插入位置为 node4 之前
- 发生线程切换
- 线程 0 修改：
  - node3->prev = node1
  - node3->next = node4
  - node1->next = node3
  - node4->prev = node3
- 发生线程切换

- 线程 1 修改:
- node2->prev = node1
- node2->next = node4
- node1->next = node2
- node4->prev = node2

期望结果: node1(1) -> node2(2) -> node3(3) -> node4(4)

实际结果: node1(1) -> node3(3) -> node2(2) -> node4(4)

结果对比: 后插入的 node2 本来应该被插入到 node3 之前, 但由于关联未修改导致失序

为了实现这种执行顺序, 手动插入元素 1, 4 后执行 `t->Fork();`, 然后分别在线程中, 先后插入 3, 2.

同时对插入元素对应位置做出修改:

```

1 void DLLList::SortedInsert(void *item, int sortKey)
2 {
3     if (first == NULL || sortKey <= first->key) {
4         Prepend(item);
5         first->key = sortKey;
6         return;
7     }
8
9     DLLElement *newItem = new DLLElement(item, sortKey),
10        *ptr = first;
11     while (ptr != NULL && ptr->key < sortKey)
12         ptr = ptr->next;
13     if (ptr == NULL) {
14         last->next = newItem,
15         newItem->prev = last,
16         last = newItem;
17     }
18     else {
19         // 找到插入位置后, 进行强制线程切换
20         if (threadTestcase == 2)
21             currentThread->Yield();
22         ptr->prev->next = newItem;
23         newItem->prev = ptr->prev;
24         newItem->next = ptr;
25         ptr->prev = newItem;
26         if (threadTestcase == 2)
27             currentThread->Yield();
28     }
29 }

```

运行结果如图所示:

```

[Event] Before insert:
      1->4 (total: 2)
[Thread 0] Inserted 3
[Thread 1] Inserted 2
      1->3->2->4 (total: 4)
      (total: 0)
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

```

**Case 3: 线程 0 向表头插入元素，线程 1 同时从表头删除元素，导致删除了非表头元素**  
此场景模拟如下：

- 初始时，链表有序非空，表头元素为 node0 (key = a) -> node1
- 线程 0 创建了一个 DLLElement 对象 node2 设置 key = b (b < a)
- 线程 0 修改了 node2->next
- 发生线程切换
- 线程 1 删除了表头元素 node0
- 线程 1 修改了 node0->next->prev 和 first 指针
- 线程 1 析构 (delete) 了 node0
- 发生线程切换
- 线程 0 修改 first 指针指向 node2

期望结果：node0(a) -> node1 -> ...

实际结果：node2(b) -> NULL

node1 -> ...

结果对比：线程 1 后执行删除操作，但删除的却不是表头的元素；同时破坏了链表的连续性。

Case 3 和 Case 1 在切换线程的位置是相同的。不同的是，Case 3 手工构造了插入的顺序。运行结果如图所示：

```

[cs182204269@mc core threads]$ ./nachos -q 2 -t 2 -n 5 -c 3
[Event] Before thread-0 insert, thread-1 remove:
      1->2 (total: 2)
[Thread 0] Inserted 0
[Thread 1] Removed first element: 1
      0->1 (total: 2)

```

**Case 4: 线程 0 和线程 1 同时删除表头元素，由于节点已被析构导致段错误**  
此场景模拟如下：

- 初始时，链表有序非空，表头元素为 node0
- 线程 0 试图删除表头节点，找到 node0
- 发生线程切换
- 线程 1 试图删除表头节点，找到 node0
- 线程 0 设置 first = first->next (NULL)
- 线程 0 将 node 0 析构
- 发生线程切换
- 线程 1 试图设置 first = first->next，但由于此时 first 为 NULL，因此发生段错误



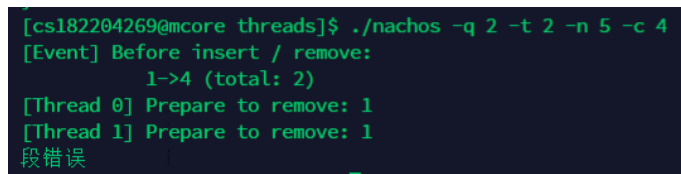
期望结果: NULL

实际结果: Segmentation Fault

对代码 Remove 函数做出的修改如下:

```
1 // remove from head of list
2 // set *keyPtr to key of the removed item
3 // return item (or NULL if list is empty)
4 void* DList::Remove(int *keyPtr)
5 {
6     if (first == NULL)
7         return NULL;
8     *keyPtr = first->key;
9     // get first remove element
10    DLLElement *removedItem = first;
11
12    // 修改 first 指向前, 强制切换线程
13    if (threadTestcase == 4) {
14        printf("Prepare to remove: %d\n", first->key);
15        currentThread->Yield();
16    }
17
18    // set first & last pointer
19    last = (first == last) ? NULL : last;
20    first = first->next;
21    if (first != NULL)
22        first->prev = NULL;
23
24    void* element = removedItem->item;
25    delete removedItem;
26    return element;
27 }
```

结果如图所示:



```
[cs182204269@mc core threads]$ ./nachos -q 2 -t 2 -n 5 -c 4
[Event] Before insert / remove:
        1->4 (total: 2)
[Thread 0] Prepare to remove: 1
[Thread 1] Prepare to remove: 1
段错误
```

### 三 实验总结与感想

通过本次实验,我们简单地阅读了 nachos 的源代码,对 nachos 的工作原理、整体情况,以及如何修改 nachos 的代码实现部分功能,有了初步的了解,并使用 C++ 实现了双向链表在 nachos 上运行测试。尤其是体验了 nachos 的线程机制,并通过实验测试了线程执行顺序的不同,对程序正确性的影响。