实验一　体验Nachos的并发程序设计

吴朱冠宇 22920182204320　胡子潇 22920182204188

谭心怡 22920182204283　卞珂 22920182204133

2021-4-1

一、实验内容

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；用C++实现双向有序链表；在nachos系统中使用你所写的链表程序并演示一些并发错误。

（一）安装nachos

将下载的nachos-linux64.tar.gz文件拷贝到实验平台的个人目录下之后，执行tar -zxvf nachos-linux64.tar.gz命令解压压缩包。在本次任务中，可单独编译线程管理部分。进入目录nachos-linux64/nachos-3.4/code/threads，先后执行命令make depend和make进行编译，可得到可执行文件nachos。执行命令./nachos运行，可观察到0、1两个线程切换的现象，说明安装成功。

```
-bash-3.2$ ./nachos
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 130, idle 0, system 130, 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...
```

图1.nachos初始运行结果

（二）实现双向有序链表

1.双向有序链表的定义

链表是一种物理结构上不连续，非顺序的数据结构,其逻辑顺序结构是以结点内的指针来确定的。双向链表是一类特殊的链表，它有两个指针，分别指向其前一个链表元素和后一个链表元素。

本实验要求实现的是一个有序双向链表，在普通双向链表结点的基础上增加一个整数域作为排序依据，将链表结点按照递增的逻辑顺序排列。下面给出本实验实现的双向链表结点类和双向链表类的定义。

（1）双向链表结点类定义（dllist.h）

双向链表结点的域包括指向前一个结点和后一个结点的两个指针prev和next、排序优先级key、指向链表元素的指针item。

```
//双向链表结点类定义（dllist.h）

class DLLElement {
public:
    // initialize a list element
    DLLElement( void *itemPtr, int sortKey );
    DLLElement *next;           // next element on list
                                // NULL if this is the last
    DLLElement *prev;           // previous element on list
                                // NULL if this is the first
    int key;                    // priority, for a sorted list
    void *item;                 // pointer to item on the list
};
```

```
// 双向链表节点类初始化定义（dillist.cc）

DLLElement::DLLElement( void *inodetr, int sortKey ) {
    next = prev = NULL;
    item = inodetr;
    key = sortKey;
}
```

（2）双向链表类定义（dllist.h）

在双向链表类中包括构造和析构函数的原型、双向链表的基本操作函数的原型、链表的首尾指针。各函数的具体实现在dllist.cc中给出。

```
//双向链表类定义（dllist.h）

class DLList {
public:
    DLList(int errType = 0);    // initialize the list
    ~DLList();                  // de-allocate the list
    void Prepend(void *item);   // add to head of list
                                // (set key = min_key-1)
    void Append(void *item);    // add to tail of list
                                // (set key = max_key+1)
    void *Remove(int *keyPtr);  // remove from head of list
                // set *keyPtr to key of the removed item
                // return item (or NULL if list is empty)
    bool IsEmpty();   // return true if list has elements

    // routines to put/get items on/off list in order
    // (sorted by key)
    void SortedInsert(void *item, int sortKey);
    // remove first item with key==sortKey
    // return NULL if no such item exists
    void *SortedRemove(int sortKey);

    void printList(DLList *list);

private:
    DLLElement *first;  // head of the list, NULL if empty
    DLLElement *last;   // last element of the list

    int errType;        // buggy behaviors to be demonstrate
                        // range from 0 to 4
};
```

```
//双向链表构造与析构函数（dllist.cc）

DLList::DLList(int Type) {
    first = last = NULL;
    this->errType = Type;
}

DLList::~DLList() {
    while (Remove(NULL) != NULL);
}
```

2.关键代码实现

H5 （1）IsEmpty() 函数实现

- 函数原型：`bool IsEmpty();`
- 函数功能：判断此时链表是否为空；
- 函数返回值：当函数为空（首尾指针相等，且等于NULL）时，返回true，其他情况下返回false(且，当链表为空单首尾指针出现不为NULL时，报错）。

```
//IsEmpty () 函数实现

bool DLList::IsEmpty() {
    if (first == NULL && last == NULL)
        return true;
    else if (first != NULL && last != NULL)
        return false;
    printf("Error! The list is empty but ");
    printf("*first or *last isn't NULL!\n");
    return false;
}
```

（2）Prepend() 函数实现

- 函数原型：`void Prepend(void *item) ;`
- 函数功能：在链表头插入结点；
- 函数参数：`*item` 为指向要插入节点的指针；
- 注意：分为当链表为空、非空两种情况。

```
//Prepend () 函数实现

void DLList::Prepend(void *item) {
    if (this->IsEmpty()) {
        DLLElement *node = new DLLElement(item, INITKEY);
        node->prev = node->next = NULL;
        first = last = node;
    }
    else {
        DLLElement *node = new DLLElement(item, first->key - 1);
        node->prev = NULL;
        node->next = first;
        first->prev = node;
        first = node;
    }
}
```

（3）Append() 函数实现

- 函数原型：`void Append(void *item) ;`
- 函数功能：在链表尾插入结点；
- 函数参数：`*item` 为指向要插入节点的指针；
- 注意：与Prepend（）函数类似，分为当链表为空、非空两种情况。

```
//Append () 函数实现

void DLList::Append(void *item) {
    if (this->IsEmpty()) {
        DLLElement *node = new DLLElement(item, INITKEY);
```

```
        node->prev = node->next = NULL;
        first = last = node;
    }
    else {
        DLLElement *node = new DLLElement(item, last->key + 1);
        node->prev = last;
        node->next = NULL;
        last->next = node;
        last = node;
    }
}
```

（4）Remove() 函数实现

- 函数原型：`void *Remove(int *keyPtr);`
- 函数功能：删除链表结点（即当前链表中keyptr指向的结点）；
- 函数参数：`keyptr` 为指向被删除结点key值的指针，即`*keyptr` 为被删除结点的key值；
- 函数返回值：返回被删除结点的指针`keyptr` ，当链表为空时，返回NULL；
- 注意：当链表为空时，首尾指针应相等，且等于NULL。

```
//Remove函数的实现（dllist.cc）

void *DLList::Remove(int *keyPtr) {
    DLLElement *element;
    void *RemovedItem;
    if (this->IsEmpty()) {
        return keyPtr = NULL;
    }
    element = first;
    *keyPtr = first->key;
    if (this->errType == 1) {
        printf("Remove error\n");
        currentThread->Yield();
    }
    RemovedItem = element->item;
    first = first->next;
```

```
    if (first == NULL) {
        last = NULL;
    }
    else {
        if (this->errType == 1) {
            printf("Remove error\n");
            currentThread->Yield();
        }
        first->prev=NULL;
    }
    return RemovedItem;
}
```

（5）SortedInsert() 函数实现

- 函数原型：`void SortedInsert(void *item, int sortKey);`
- 函数功能：按从小到大的顺序插入链表结点；
- 函数参数：`*item` 为指向插入结点的指针，`sortKey` 为当前指针指向的结点key值；
- 注意：分为四种情况（空链表、首插、尾插、中间位置插入）进行讨论。

```
//SortedInsert()函数的实现

void DLList::SortedInsert(void *item, int sortKey) {
    if (this->IsEmpty()) {
        DLLElement *node = new DLLElement(item, sortKey);
        first = node;
        if (errType == 2) {
            printf("SortedInsert error, first != last\n");
            currentThread->Yield();
        }
        node->prev = node->next = NULL;
        last = node;
    }
    else {
        DLLElement *node = new DLLElement(item,sortKey);
        DLLElement *inst = first;
```

```
        while (inst != NULL && sortKey >= inst->key) {
            inst = inst->next;
        }
        if (errType == 3) {
            printf("SortedInsert error, the postion lost\n");
            currentThread->Yield();
        }
        if (inst == NULL) { //在表尾插入
            node->prev = last;
            node->next = NULL;
            last->next = node;
            last=node;
        }
        else if (inst == first) { //在表头插入
            node->prev = NULL;
            node->next = first;
            first->prev = node;
            first = node;
        }
        else {
            node->prev = inst->prev;
            if (errType == 4) {
                printf("SortedInsert error, sorting error\n");
                currentThread->Yield();
            }

            inst->prev->next = node;
            inst->prev = node;
            node->next = inst;
        }
    }
}
```

（6）SortedRemove() 函数实现

- 函数原型：`void SortedRemove(int sortKey);`
- 函数功能：按从小到大的顺序删除链表结点；
- 函数参数：`sortKey` 为当前被删除的结点的key值；

- 函数返回值：返回`*ReturnItem`，即接下来需要删除的结点的指针；
- 注意：与SortedInsert（）函数相似，同样分为四种情况（空链表、首删、尾删、中间位置删除）进行讨论。

```
//SortedRemove () 函数实现

void *DLList::SortedRemove(int sortKey) {
    void *ReturnItem;
    if (this->IsEmpty()) {
        return NULL;
    }
    else {
        DLLElement *delt = first;
        while (delt != NULL && sortKey != delt->key) {
            delt=delt->next;
        }
        if (delt == NULL) {
            return NULL;
        }
        else {
            ReturnItem = delt->item;
            if (delt == first) {
                int *key;
                this->Remove(key);
            }
            else if (delt == last) {
                last = last->prev;
                last->next = NULL;
                if (last == NULL) {
                    first = NULL;
                }
            }
            else {
                delt->prev->next = delt->next;
                delt->next->prev = delt->prev;
            }
            return ReturnItem;
        }
```

```
    }
}
```

（7）dllist-driver.cc实现

dllist-driver.cc中定义供线程操作双向链表的函数接口，线程只能通过这两个函数对双向链表进行操作。

① InsertList() 函数实现

- 函数原型：`void InsertList(int threadNum, int N, DLList *list);`
- 函数功能：向双向链表中插入N个结点，每个结点的key值为[1,100]的随机数；
- 函数参数：`threadNum` 为线程号，`N` 为插入的结点数量，`list` 为双向链表的类指针；
- 实现思路：调用`SortedInsert()` 函数，以当前时间为种子值，每次生成一个随机数作为结点key值；
- 注意：为了避免伪随机数的影响（n个线程可能在极短时间内陆续开启，而此时作为种子值的时间并未改变），将种子值改为当前时间与线程号之和。这样既可以让每次实验生成的随机数不同，又可以让单次实验中每个线程生成的随机数不同。

```
// InsertList函数实现

void InsertList(int threadNum, int N, DLList *list) {
    RandomInit(unsigned(time(0) + threadNum));
    for (int i = 0; i < N; i++) {
        void *item;
        int key = Random() % 100;
        int *items = new int[1];
        items[0] = -key;
        item = items;
        list->SortedInsert(item, key);
        printf("Thread %d: Inserted key %2d | ",
               threadNum, key);
        list->printList(list);
    }
}
```

H6 ②RemoveList() 函数实现

- 函数原型：`void RemoveList(int threadNum, int N, DLList *list);`
- 函数功能：从头开始删除双向链表中N个结点；
- 函数参数：`threadNum` 为线程号，`N` 为插入的结点数量，`list` 为双向链表的类指针；
- 函数返回值：调用N次`Romove()` 函数。

```
void RemoveList(int threadNum, int N, DLList *list) {
    for (int i = 0; i < N; i++) {
        int key;
        list->Remove(&key);
        printf("Thread %d: Removed  key %2d | ",
               threadNum, key);
        list->printList(list);
    }
}
```

其余函数在本次实验中未被使用，具体实现可见源代码文件，这里不做详细分析。

3.演示正常情况下的程序运行

H5 （1）设计命令行参数，修改main.cc

命令格式：`./nachos [-q testnum] [-t threadNum] [-n insCnt] [-e errType]`，其中，四个参数依次为测试类型（测试类型为1时，使用默认，测试类型为2时，使用即将输入的参数）、线程数、操作节点数、错误类型。

在main.cc中响应做如下修改：

- 增加提示输入参数格式的help指令
- 声明相关参数的全局变量
- 增加命令行参数处理

```
//命令行参数处理修改

#ifdef THREADS
    if (argc == 2 && !strcmp(argv[1], "--help")) {
        printf("<usage>: ./nachos [-q testnum] [-t threadNum]);
        printf("[-n insCnt] [-e errType]\n\n");
    }
    else {
        for (argc--, argv++; argc > 0;
             argc -= argCount, argv += argCount) {
            argCount = 1;
            switch (argv[0][1]) {
                case 'q':
                    testnum = atoi(argv[1]);
                    argCount++;
                    break;
                case 't':
                    threadNum = atoi(argv[1]);
                    argCount++;
                    break;
                case 'n':
                    insCnt = atoi(argv[1]);
                    argCount++;
                    break;
                case 'e':
                    errType = atoi(argv[1]);
                    if (errType < 0 || errType > 4) {
                        printf("Error! errType should ");
                        printf("in [0, 4].\n");
                        ASSERT(errType >= 0 && errType <= 4);
                    }
                    argCount++;
                    break;
                default:
                    testnum = 1;
                    break;
            }
        }
```

```
        ThreadTest();
    }
    // Hello myHello;
    // myHello.Hi();

#endif
```

H5 （2）设计并发程序运行方式，修改threadtest.cc

在`threadtest.cc`中，需要规定每个线程对双向链表的操作并建立规定数量的线程，这里简单起见，这里规定每个线程先顺序插入N个结点然后再顺序删除N个结点。对此，需要做如下修改：

H6 ①在全局变量中引入命令行参数并创建双向链表

```
// testnum is set in main.cc
int testnum = 1, threadNum = 1, insCnt = 1, errType = 0;
DLList *dllist;
```

H6 ②编写ThreadTest2()，初始化dllist，创建（fork）threadNum个线程，每个线程执行函数DLListThread()

```
void ThreadTest2() {
    DEBUG('t', "Entering ThreadTest2");
    dllist = new DLList(errType);
    for (int i = 1; i < threadNum; i++) {
        Thread *t = new Thread("forker thread");
        t->Fork(DLListThread, i);
    }
    DLListThread(threadNum);
}
```

H6 ③定义 DLListThread()，其规定了每个线程执行的操作

```
void DLListThread(int t) {
    InsertList(t, insCnt, dllist);
    RemoveList(t, insCnt, dllist);
}
```

（3）修改Makefile.common文件，重新编译nachos

```
PROGRAM = nachos

THREAD_H =../threads/copyright.h\
    # 略
    ../threads/dillist.h

THREAD_C =../threads/main.cc\
    # 略
    ../threads/dillist.cc\
    ../threads/dillist-driver.cc

THREAD_S = ../threads/switch.s

THREAD_O =main.o list.o scheduler.o synch.o synchlist.o \
    system.o thread.o utility.o threadtest.o interrupt.o \
    stats.o sysdep.o timer.o dillist.o dillist-driver.o

```

H5 （4）运行程序，观察正常情况下的程序运行

```
[cs182204320@mcore threads]$ ./nachos -q 2 -t 2 -n 3
Thread 2: Inserted key 86 | Current List: 86

Thread 2: Inserted key 41 | Current List: 41 86

Thread 2: Inserted key 15 | Current List: 15 41 86

Thread 2: Removed  key 15 |Current List: 41 86

Thread 2: Removed  key 41 |Current List: 86

Thread 2: Removed  key 86 |Current List:

Thread 1: Inserted key 73 | Current List: 73

Thread 1: Inserted key 47 | Current List: 47 73

Thread 1: Inserted key 31 | Current List: 31 47 73

Thread 1: Removed  key 31 |Current List: 47 73

Thread 1: Removed  key 47 |Current List: 73

Thread 1: Removed  key 73 |Current List:

No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 30, idle 0, system 30, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
```

（三）并发错误演示与分析

错误一

- 执行操作

插入过程中发生的并发错误，在第一个元素插入前发生进程切换。

```
void DLList::SortedInsert(void *item, int sortKey) {
        if (errType == 1) {     // (错误一) 在应当插入元素之前就切换了线程
            printf("SortedInsert error, Switching before inserting\n"); //可能会造成由线程X插入的元素之后被线程Y删除，而不能被自己删除
            currentThread->Yield();
        }
    if (this->IsEmpty()) {
        DLLElement *node = new DLLElement(item, sortKey);       //创建新的待插入节点
        first = node;
        node->prev = node->next = NULL;
        last = node;
```

运行查看可能出现的问题，并发三个线程，每个线程的操作节点个数为4：

- 运行结果

程序正常结束，但因进程切换导致插入、删除操作混乱，线程删除其他线程插入的元素。

```
[cs182204133@mcore threads]$ ./nachos -q 2 -t 3 -n 4 -e 1
Thread 3: Inserted key  3  | SortedInsert error, Switching before inserting
Thread 1: Inserted key 30  | SortedInsert error, Switching before inserting
Thread 2: Inserted key  2  | SortedInsert error, Switching before inserting
Current List:  3

Thread 3: Inserted key 35  | SortedInsert error, Switching before inserting
Current List:  3 30

Thread 1: Inserted key 64  | SortedInsert error, Switching before inserting
Current List:  2  3 30

Thread 2: Inserted key 85  | SortedInsert error, Switching before inserting
Current List:  2  3 30 35

Thread 3: Inserted key  0  | SortedInsert error, Switching before inserting
Current List:  2  3 30 35 64
```

- 错误分析

可见第三行Thread 1向链表中插入元素30，但因为在实际运行过程中线程未进行插入操作就退出，所以链表并无元素30，而是执行第一次被打断的操作：Thread3插入元素3；第六行Thread 3向链表插入元素，因实际运行中线程未进行插入操作就退出，之后**Thread 1获得执行权，执行上次被打断的插入操作：向链表插入30**，使得链表中新增元素30。

```
Thread 3: Removed  key  0 |Current List:  2  3 30 35 57 64 69 82 85
Thread 3: Removed  key  2 |Current List:  3 30 35 57 64 69 82 85
Thread 3: Removed  key  3 |Current List: 30 35 57 64 69 82 85
Thread 3: Removed  key 30 |Current List: 35 57 64 69 82 85
Current List: 15 35 57 64 69 82 85
```

链表元素删除时，**Thread 3删除了30元素**。但对比上图插入元素过程，**调用插入30的是Thread 1**。由于线程在元素删除时并无混乱，而线程插入时错误切换，最终导致每个线程都有可能删除并未插入的节点。

错误二

- 执行操作

链表为空的情况下，插入节点后立刻切换了进程。

```
if (this->IsEmpty()) {
    DLLElement *node = new DLLElement(item, sortKey);  //创建新的待插入节点
    first = node;
    node->prev = node->next = NULL;
    last = node;
    if (errType == 2) { // (错误二) 链表为空的情况下，插入该节点之后立即就切换了进程,同样会造成
        printf("SortedInsert error, insert one and switch immediately \n"); //由线程X插入的元素之后被线程Y删除，而不能被自己删除
        currentThread->Yield();
    }
}
```

运行查看可能出现的问题，并发三个线程，每个线程的操作节点个数为4：

- 运行结果

程序正常结束。线程删除的顺序出现了混乱，每个线程插入和删除的元素并不相同。

```
[cs1822041133@mcore threads]$ ./nachos -q 2 -t 3 -n 4 -e 2
Thread 3: Inserted key 61  | SortedInsert error, insert one and switch immediately
Thread 1: Inserted key 82  | Current List: 61 82
Thread 1: Inserted key 48  | Current List: 48 61 82
Thread 1: Inserted key 42  | Current List: 42 48 61 82
Thread 1: Inserted key 13  | Current List: 13 42 48 61 82
Thread 1: Removed  key 13  |Current List: 42 48 61 82
Thread 1: Removed  key 42  |Current List: 48 61 82
Thread 1: Removed  key 48  |Current List: 61 82
Thread 1: Removed  key 61  |Current List: 82
```

- 错误分析

可见一开始Thread 3进行了61的插入，但由于链表为空时插入节点后进程切换，其后Thread 1正常执行4个链表插入操作。此时链表中已有5个元素，而后Thread 1进行4个链表删除操作，因删除操作是从头节点开始，Thread 1删除了Thread 3插入的节点61，导致线程删除和插入的不统一。

错误三

- 执行操作

链表不空，且需在表尾进行插入，但是只将node->prev连接，而并未修改其他连接，导致节点插入无效。

```
if (inst == NULL) { //在表尾插入
    node->prev = last;
    if (errType == 3) {    //（错误三）在表尾插入时，只修改了node->prev，而没有修改其他指针
    printf("SortedInsert error, pointer error while inserting end of list\n");
    currentThread->Yield();
    }
    node->next = NULL;
    last->next = node;
    last=node;
}
```

运行查看可能出现的问题，并发三个线程，每个线程的操作节点个数为4：

- 运行结果

程序正常结束。链表失序，线程删除的顺序出现了混乱，每个线程插入和删除的元素并不相同。

```
Thread 3: Inserted key 92  | SortedInsert error, pointer error while inserting end of list
Thread 1: Inserted key 81  | SortedInsert error, pointer error while inserting end of list
Thread 2: Inserted key 64  | Current List:  4 64 77

Thread 2: Inserted key 43  | Current List:  4 43 64 77

Thread 2: Inserted key 80  | SortedInsert error, pointer error while inserting end of list
Current List:  4 43 64 77 92

Thread 3: Inserted key 30  | Current List:  4 30 43 64 77 92

Thread 3: Removed  key  4  |Current List: 30 43 64 77 92

Thread 3: Removed  key 30  |Current List: 43 64 77 92

Thread 3: Removed  key 43  |Current List: 64 77 92

Thread 3: Removed  key 64  |Current List: 77 92

Current List: 77 92 81
```

```
Thread 2: Inserted key 10  | Current List: 10 92 81 80

Thread 2: Removed  key 10  |Current List: 92 81 80

Thread 2: Removed  key 92  |Current List: 81 80

Thread 2: Removed  key 81  |Current List: 80

Thread 2: Removed  key 80  |Current List:
```

- 错误分析

Thread 2试图在表尾插入92，Thread 1在表尾插入81，二者都在未进行插入时就进行线程切换，之后在线程切换中Thread 2先实现在链表的last节点后插入元素，链表尾节点last的key=92。在此之后Thread 3才得以真正在链表中插入81，因各线程公用双向链表，导致Thread 3在现在链表的尾节点（key=92）后插入81，导致链表失序。

而在删除过程中，Thread 2最后删除链表中仅剩的4个节点，因此删除了之前在表尾插入的81，出现线程删除不是自己插入的元素。

错误四

- 执行操作

链表不空，且需在表头进行插入，但在插入操作实际进行之前就切换进程。

```
else if (inst == first) { //在表头插入

    if (errType == 4) {                                  //（错误四）此时链表不空，而且要在表头进行插入
    printf("SortedInsert error, switch before insert at the head of list\n");   //但是还未插入该节点就切换了进程
    currentThread->Yield();
    }
    node->prev = NULL;
    node->next = first;
```

运行查看可能出现的问题，并发三个线程，每个线程的操作节点个数为4：

- 运行结果

程序正常结束。链表失序,且线程删除的顺序出现了混乱，每个线程插入和删除的元素并不相同。

```
Thread 4: Inserted key 91  | SortedInsert error, switch before insert at the head of list
Thread 1: Inserted key 44  | SortedInsert error, switch before insert at the head of list
Thread 2: Inserted key 62  | SortedInsert error, switch before insert at the head of list
Thread 3: Inserted key 68  | SortedInsert error, switch before insert at the head of list
Current List: 91 93

Thread 4: Inserted key 49  | SortedInsert error, switch before insert at the head of list
Current List: 44 91 93

Thread 1: Inserted key 33  | SortedInsert error, switch before insert at the head of list
Current List: 62 44 91 93

Thread 4: Removed  key  0 |Current List: 67 62 44 67 91 91 93 96

Thread 4: Removed  key 67 |Current List: 62 44 67 91 91 93 96

Thread 4: Removed  key 62 |Current List: 44 67 91 91 93 96

Thread 4: Removed  key 44 |Current List: 67 91 91 93 96

Current List: 21 67 91 91 93 96
```

- 错误分析

Thread 1试图在表头插入44，Thread 2在表头插入62，二者都在未进行插入时就进行线程切换，之后在线程切换中Thread 1先实现在链表的first节点前插入元素，链表头节点first的key=44。在此之后Thread 2才得以真正在链表中插入62，因各线程公用双向链表，导致Thread 2在现在链表的头节点（key=44）前插入62，导致链表失序。

而在删除过程中，Thread 4删除链表中仅剩的4个节点，因此删除了之前在表头插入的44，出现线程删除不是自己插入的元素。

错误五

- 执行操作：

  此时链表不空，而且要在表头进行插入，但是只将node->next连接，而并未连接first->prev

```
else if (inst == first) { //在表头插入

    if (errType == 4) {                                  //（错误四）此时链表不空，而且要在表头进行插入
    printf("SortedInsert error, switch before insert at the head of list\n");   //但是还未插入该节点就切换了
    currentThread->Yield();
}
    node->prev = NULL;
    node->next = first;
    if (errType == 5) {                                  //（错误五）此时链表不空，而且要在表头进行插入
    printf("SortedInsert error, pointer error while inserting head of list\n");   //但是只将node->next连接,
    currentThread->Yield();
}
    first->prev = node;
    first = node;
}
```

  运行查看可能出现的问题，并发三个线程，每个线程的操作节点个数为6：

  - 运行结果

```
[cs182204283@mcore threads]$ ./nachos -q 2 -t 3 -n 6 -e 5
Thread 3: Inserted key 89 | Current List: 89

Thread 3: Inserted key  7 | SortedInsert error, pointer error while inserting head of list
Thread 1: Inserted key  7 | SortedInsert error, pointer error while inserting head of list
Thread 2: Inserted key 29 | SortedInsert error, pointer error while inserting head of list
Current List:  7 89

Thread 3: Inserted key 19 | Current List:  7 19 89

Thread 3: Inserted key 40 | Current List:  7 19 40 89

Thread 3: Inserted key 79 | Current List:  7 19 40 79 89

Thread 3: Inserted key 31 | Current List:  7 19 31 40 79 89

Thread 3: Removed  key  7 | Current List: 19 31 40 79 89

Thread 3: Removed  key 19 | Current List: 31 40 79 89

Thread 3: Removed  key 31 | Current List: 40 79 89

Thread 3: Removed  key 40 | Current List: 79 89

Thread 3: Removed  key 79 | Current List: 89

Thread 3: Removed  key 89 | Current List:
段错误
```

  发现出现了段错误。

- 错误分析：

在该例中，共有三个线程，每个线程都要向链表中插入6个节点，然后再删除掉6个节点。

首先，线程0执行插入，第一个数值为89，然后插入第二个数值，该数值为7，小于89，所以满足“链表不为空且当前要在链表头部插入”这一线程切换条件，当待插入节点的尾指针已经指向当前头节点后，立即切换进程。随后，线程1获得资源，开始执行，线程一想要插入的数值为7，同样满足了上面的线程切换条件，于是线程再次切换。这时，轮到了线程2占用资源，而线程2想要插入的数值为29，同样满足了上面的线程切换条件，于是线程再次切换。此时，线程0终于再次获得了资源，得以继续运行。（**请注意，这时由线程0、线程1、线程2这三个线程创建的三个待插入表头的节点，他们的尾指针均指向了当前的first头指针。**）

然后线程0进行正常的插入6个节点，且没有再次遇到需要插入表头的情况，然后删除六个节点，至此线程0执行完毕，资源释放。然后线程1和线程2开始资源的竞争，二者其中之一得到了资源，开始运行，无论这二者哪一个得到了资源，都是执行在表头插入节点的操作，即“first->prev = node;”这一句代码，但是由于之前链表中元素已经被清除，因此first指针也被释放，所以现在实际上是在调用一个已经被释放了的指针，所以会出现段错误。

错误六

- 执行操作：

在链表中间插入时，还未成功插入该节点就切换了进程，只指定了 node->prev而未指定之后指针的操作。

```
else {                                          //执行普通的插入操作
    node->prev = inst->prev;
    if (errType == 6) {                         // (错误六) 在链表中间插入时，还未成功插入该节点就切换了
        printf("SortedInsert error, pointer error while inserting middle of list\n");  //只指定了 node->pre
        currentThread->Yield();
    }

    inst->prev->next = node;
    inst->prev = node;
    node->next = inst;
}
```

运行查看可能出现的问题，并发两线程，每个线程的操作节点个数为6个。

- 运行结果

```
[cs182204283@mcore threads]$ ./nachos -q 2 -t 2 -n 6 -e 6
Thread 2: Inserted key 11 | Current List: 11

Thread 2: Inserted key 43 | Current List: 11 43

Thread 2: Inserted key 15 | SortedInsert error, pointer error while inserting middle of list
Thread 1: Inserted key 78 | Current List: 11 43 78

Thread 1: Inserted key 17 | SortedInsert error, pointer error while inserting middle of list
Current List: 11 15 43 78

Thread 2: Inserted key 11 | SortedInsert error, pointer error while inserting middle of list
Current List: 11 15 17 43 78

Thread 1: Inserted key 38 | SortedInsert error, pointer error while inserting middle of list
Current List: 11 11 15 17 43 78

Thread 2: Inserted key 73 | SortedInsert error, pointer error while inserting middle of list
Current List: 11 11 15 17 38 43 78

Thread 1: Inserted key 69 | SortedInsert error, pointer error while inserting middle of list
Current List: 11 11 15 17 38 43 73 78

Thread 2: Inserted key 24 | SortedInsert error, pointer error while inserting middle of list
Current List: 11 11 15 17 38 43 73 69 78

Thread 1: Inserted key  8 | Current List:  8 11 11 15 17 38 43 73 69 78

Thread 1: Inserted key 87 | Current List:  8 11 11 15 17 38 43 73 69 78 87

Thread 1: Removed  key  8 | Current List: 11 11 15 17 38 43 73 69 78 87

Thread 1: Removed  key 11 | Current List: 11 15 17 38 43 73 69 78 87

Thread 1: Removed  key 11 | Current List: 15 17 38 43 73 69 78 87

Thread 1: Removed  key 15 | Current List: 17 38 43 73 69 78 87

Thread 1: Removed  key 17 | Current List: 38 43 73 69 78 87

Thread 1: Removed  key 38 | Current List: 43 73 69 78 87
段错误
```

发现出现了删除时线程可能无法删除自己插入的元素，而是删除了其它线程插入的元素；同时还出现了段错误。

- 错误分析：

当在链表中间插入一个节点的时候，只将待插入节点的头指针指向了插入位置节点的前一个节点，然后就切换了进程，相当于待插入节点只有一端连在了链表之上，然后就暂停了插入。其他进程之后占用了资源，然后对该链表进行修改，但是其他进程在修改的过程中，可能会对之前还未插入完成的位置进行修改，而这样就可能会造成链表中已经插入的节点可能丢失，或是最后切换回未完成插入操作的线程后发现相关节点已被删除，从而产生段错误。

错误七

- 执行操作：

  找出了要删除的元素，但还并未将其删除就切换了线程：

```
printf("%2d |", first->key);                    //输出要删除的元素的值
if (this->errType == 7) {                       //（错误七）找出了要删除的元素，但还并未将其删除就切换了线程
    printf("Remove error 1\n");
    currentThread->Yield();
}
RemovedItem = element->item;
first = first->next;                            //删除操作
```

  运行查看可能出现的问题，并发线程两个，每个线程的操作节点个数为6个：

  - 运行结果

```
[cs182204283@mcore threads]$ ./nachos -q 2 -t 2 -n 6 -e 7
Thread 2: Inserted key 16 | Current List: 16

Thread 2: Inserted key  3 | Current List:  3 16

Thread 2: Inserted key 46 | Current List:  3 16 46

Thread 2: Inserted key 26 | Current List:  3 16 26 46

Thread 2: Inserted key 83 | Current List:  3 16 26 46 83

Thread 2: Inserted key 83 | Current List:  3 16 26 46 83 83

Thread 2: Removed  key  3 |Remove error 1
Thread 1: Inserted key 82 | Current List:  3 16 26 46 82 83 83

Thread 1: Inserted key 89 | Current List:  3 16 26 46 82 83 83 89

Thread 1: Inserted key 33 | Current List:  3 16 26 33 46 82 83 83 89

Thread 1: Inserted key 80 | Current List:  3 16 26 33 46 80 82 83 83 89

Thread 1: Inserted key 97 | Current List:  3 16 26 33 46 80 82 83 83 89 97

Thread 1: Inserted key 54 | Current List:  3 16 26 33 46 54 80 82 83 83 89 97
```

```
Thread 1: Removed  key  3 |Remove error 1
Current List: 16 26 33 46 54 80 82 83 83 89 97

Thread 2: Removed  key 16 |Remove error 1
Current List: 26 33 46 54 80 82 83 83 89 97

Thread 1: Removed  key 26 |Remove error 1
Current List: 33 46 54 80 82 83 83 89 97

Thread 2: Removed  key 33 |Remove error 1
Current List: 46 54 80 82 83 83 89 97

Thread 1: Removed  key 46 |Remove error 1
Current List: 54 80 82 83 83 89 97

Thread 2: Removed  key 54 |Remove error 1
Current List: 80 82 83 83 89 97

Thread 1: Removed  key 80 |Remove error 1
Current List: 82 83 83 89 97

Thread 2: Removed  key 82 |Remove error 1
Current List: 83 83 89 97

Thread 1: Removed  key 83 |Remove error 1
Current List: 83 89 97

Thread 2: Removed  key 83 |Remove error 1
Current List: 89 97

Thread 1: Removed  key 89 |Remove error 1
Current List: 97

Current List:
```

观察实验结果可以发现，显示的每个线程删除的节点值总是上一个被删除的节点的值，而非他实际删除的节点的值，同时在删除的过程中，并非是一个线程全部删除完6个元素之后再切换，而是两个线程之间不断切换进行删除，两个线程删除的节点也不一定是自己之前插入的节点值。

- 错误分析：

比如，3这个节点，开始时应当是线程2去删除，但由于遇到了线程切换，因此暂时不能删除，随后切换至线程1，线程1顺利完成插入操作，当它想去删除3节点时，也遇到了线程切换，此时线程2就可以将节点3删除掉，之后线程2再次遇到线程切换，线程1就开始删除，但此时线程1的删除已经不再是3元素，而是它的下一个元素。

这是因为，我们将切换的位置放在了删除节点具体操作之前，此时线程已经得知了目前要进行删除的元素的值，但就立即切换至另一个线程，该线程首先完成插入操作，之后再执行删除操作，这时第二个线程也由于发生错误而进行线程切换，于是资源重新回到了线程手中。于是现在线程一顺利删除掉第一个元素，但进行第二个元素删除操作时又发生了错误，于是线程再次切换至线程二，不断反复，直至完成所有元素的删除操作。这样一来，每个线程就不能正好删除自己

插入的元素，而可能会去删除另外一个线程所插入的元素；而且，线程实际删除的元素也可能与切换之前想要删除的元素不一样，因为头节点也可能在线程切换之间被修改。

错误八

- 执行操作：

已经删除了元素，但是还未将头指针first的前驱节点改为NULL，并且还未返回删除元素的数值

```
RemovedItem = element->item;
first = first->next;                          //删除操作
if (first == NULL) {
    last = NULL;
}
else {
    if (this->errType == 8) {             //（错误八）已经删除了元素，但是还未返回删除元素的数值，此时切换
        printf("Remove error 2\n");
        currentThread->Yield();
    }
    first->prev=NULL;
}
return RemovedItem;                       //返回刚刚返回的值
```

运行查看可能出现的问题，并发线程两个，每个线程的操作节点个数为6个：

- 运行结果

```
[cs182204283@mcore threads]$ ./nachos -q 2 -t 2 -n 3 -e 8
Thread 2: Inserted key 16 | Current List: 16

Thread 2: Inserted key 31 | Current List: 16 31

Thread 2: Inserted key 82 | Current List: 16 31 82

Thread 2: Removed  key 16 |Remove error 2
Thread 1: Inserted key 97 | Current List: 31 82 97

Thread 1: Inserted key 34 | Current List: 31 34 82 97

Thread 1: Inserted key 24 | Current List: 24 31 34 82 97

Thread 1: Removed  key 24 |Remove error 2
Current List: 31 34 82 97

Thread 2: Removed  key 31 |Remove error 2
Current List: 34 82 97

Thread 1: Removed  key 34 |Remove error 2
Current List: 82 97

Thread 2: Removed  key 82 |Remove error 2
Current List: 97

Thread 1: Removed  key 97 |Current List:
段错误
```

线程二首先占用了资源，然后进行插入操作，之后开始删除，在删除第一个的最后进行线程切换，切换至第一个之后，线程一同样要进行上述的插入和删除操作在这个错误中，就不会出现输出的删除元素的值与实际删除的不一样的问题，但是会出现段错误。

- 错误分析：

线程一将链表中最后一个元素删除后，**线程二接管CPU，继续执行first -> prev = NULL**。但由于此时链表已为空，即first =NULL。first -> prev = NULL执行出现错误。

三、实验总结

此次实验最开始的时候，我们是很迷茫的，并不知道自己要做什么。我们查阅了不少中文资料，却大多不知所云。最终，我们选择了一字一句认真阅读翻译英文文档。过程虽然艰辛，但是我们逐渐明白了任务是什么，并且有了一个大体的思路。这个小插曲告诉我们，学习这件事情没有捷径可以走，唯有静下心来踏踏实实钻研，才能收获知识与能力。

本次实验重点在于体验Nachos的线程系统，但在实现双向有序链表的过程我们也收获颇丰。尽管我们在数据结构课上早已实现过双向链表，但此次实验需要我们将各种函数封装好，并向外提供API。在设计操作系统这么一个浩大的工程中，我们必须要尽可能分层次实现，高层隐藏底层的细节。这一思想，在此次实验中有了初步的体现。同时，「时间+线程号」作为随机种子的想法，让我们实现了「真随机数」，也是很有成就感的一件事情。

然后就是分析程序在并发条件下运行的部分。本次实验让我们感受到并发情况下程序运行的不同之处。由于并发的存在，在单线程条件下能正常运行的程序却出现了乱序、结点丢失、链表结构被破坏甚至还会因为插入操作进行至一半而被打断导致指针访问以释放的内存空间使得程序异常终止。这些现象都让我们认识到了在多线程情况下「互斥」和「同步」的重要性。

最后就是这种小组合作完成的实验的形式也是给这次实验带来了一些挑战。在最开始，我们就定下了通过GitHub合作的方案。我们学习了git有关的各种操作，为每次实验创建了一个分支；大家修改完代码后直接push到项目中即可，并且可以方便地查看历史版本，大大提高了效率。

附录：小组分工情况

任务编号	任务内容	完成
任务1	dillist.cc 实现	吴朱冠宇、胡子潇
	dillist-driver.cc 实现	谭心怡
任务2	main.cc、threadtest.cc、dillist.cc 修改	卞珂
	并发错误发现与分析	全体成员
实验报告	并发错误分析部分	谭心怡、卞珂
	其他部分	吴朱冠宇、胡子潇