操作系统实验一：体验 Nachos 下的并
发程序设计

姓名	学号
陈锦涛	22920202204542

实验目的

- 熟悉 nachos 系统，了解其目录结构
- 学习实验 Mikefile 工具
- 初步体验 Nachos 下的并发程序设计

实验要求

- 安装 nachos 并阅读相关代码
- 尝试修改 Nacho，s 源代码
- 实现双向有序链表。撰写 $dllist.h$, $dllist.cc$, $dllist-driver$ 程序。
- 体验 Nachos 线程系统。

实验内容

1. 在 `nachos-3.4/code/threads` 目录下完成 `dllist.h`， `dllist.cc` ，`dllist-driver.cc` 文件的编写。
2. 修改 `Makefile.commom` 中的 THREAD_H、THREAD_C、THREAD_O。
3. 修改 `main.cc`， `threadtest.cc`，通过操作双向链表展现多线程并发所引发的问题。

实验设计

1. 修改 Makefile 文件

为使 `make` 命令能够正常工作，我们需要向 *Makefile.common* 文件中相关的部分加入我们新增的 `dllist.cc`，`dllist-driver.cc` ，`dllist.h` 文件信息与这些文件在编译过程中产生的中间文件信息，从而使得 `make depend` 命令能够正确判断源文件间的依赖关系进而支持 `make` 正常编译链接出目标文件。

修改如图：

```
THREAD_H =../threads/copyright.h\
    ../threads/list.h\
    ../threads/scheduler.h\
    ../threads/synch.h \
    ../threads/synchlist.h\
    ../threads/system.h\
    ../threads/thread.h\
    ../threads/utility.h\
    ../threads/hello.h\
    ../threads/dllist.h\
    ../machine/interrupt.h\
    ../machine/sysdep.h\
    ../machine/stats.h\
    ../machine/timer.h
```

```
THREAD_C =../threads/main.cc\
    ../threads/list.cc\
    ../threads/scheduler.cc\
    ../threads/synch.cc \
    ../threads/synchlist.cc\
    ../threads/system.cc\
    ../threads/thread.cc\
    ../threads/utility.cc\
    ../threads/threadtest.cc\
    ../threads/dllist.cc\
    ../threads/dllist-driver.cc\
    ../threads/hello.cc\
    ../machine/interrupt.cc\
    ../machine/sysdep.cc\
    ../machine/stats.cc\
    ../machine/timer.cc
```

```
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 hello.o \
dllist.o dllist-driver.o
```

2. 双向链表的实现

- dllist.h

  与实验指导的内容一致，为了更好的测试，添加了一些新的函数如下。

```C++
#ifndef DLIST_H
#define DLIST_H

class DLLElement{
        public:
        DLLElement(void *itemPtr, int sortKey); // initialize a list

        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
};

class DLList{
        public:
        DLList();  // initialize the list
        ~DLList(); // de-allocate the list

        void Prepend(void *item);  // add to head of list (set key =
        void Append(void *item);   // add to tail of list (set key =
        void *Remove(int *keyPtr); // remove from head of list
        // return true if list has elements
        bool IsEmpty();
        // routines to put/get items on/off list in order (sorted by
        void SortedInsert(void *item, int sortKey);
        void SortedInsert2(void *item, int sortKey);
        void SortedInsert3(void *item, int sortKey);
        // remove first item with key==sortKey
        void *SortedRemove(int sortKey);
        //求链表长度
        int CountList();
        //第n个结点的key值
```

```
        int FoundKey(int n);
        // 打印整个链表 因为线程的切换会导致两次连续打印 所以区分是插入时打印还
        void ShowList();
        void ShowList1();
        void ShowList2();
        //逆序打印链表
        void ReverseShowList();
        private:
        DLLElement *first; // head of the list, NULL if empty
        DLLElement *last;  // last element of the list, NULL if empty
};
#endif
```

- dllist.cc

实现 `dllist.h` 里面的所有函数，下面仅展示重要的函数内容。

C++

```
void DLList::SortedInsert(void *item, int sortKey)
{ // 按顺序插入
    DLLElement *p = first;
    DLLElement *_new = new DLLElement(item, sortKey);
    if (first == NULL) {
        first = _new;
        last = _new;
        return;
    }
    if (sortKey < first->key) { // 插入第一个
        _new->next = first;
        first->prev = _new;
        first = _new;
        return;
    }
    while (p->next != NULL) { // 插入中间
        p = p->next;
        if (sortKey < p->key) {
            _new->next = p;
            _new->prev = p->prev;
            p->prev->next = _new;
            p->prev = _new;
            return;
        }
    }
    if (sortKey >= p->key) { // 插入最后一个
```

```
        _new->prev = p;
        p->next = _new;
        last = _new;
    }
};

void DLList::SortedInsert2(void *item, int sortKey)
{ //覆盖和断链测试
    DLLElement *p = first;
    DLLElement *_new = new DLLElement(item, sortKey);
    if (first == NULL) {
        first = _new;
        last = _new;
        return;
    }
    if (sortKey < first->key) { // 插入第一个
        _new->next = first;
        currentThread->Yield();
        first->prev = _new;
        first = _new;
        return;
    }
    while (p->next != NULL) { // 插入中间
        p = p->next;
        if (sortKey < p->key) {
            _new->next = p;
            currentThread->Yield();
            _new->prev = p->prev;
            p->prev->next = _new;
            p->prev = _new;
            return;
        }
    }
    if (sortKey >= p->key) { // 插入最后一个
        _new->prev = p;
        currentThread->Yield();
        p->next = _new;
        last = _new;
    }
};

void DLList::SortedInsert3(void *item, int sortKey)
{ //乱序插入测试
    DLLElement *p = first;
```

```
    DLLElement *_new = new DLLElement(item, sortKey);
    if (first == NULL) {
        first = _new;
        last = _new;
        return;
    }
    if (sortKey < first->key) { // 插入第一个
        _new->next = first;
        first->prev = _new;
        first = _new;
        return;
    }
    while (p->next != NULL) { // 插入中间
        p = p->next;
        if (sortKey < p->key) {
            _new->next = p;
            _new->prev = p->prev;
            currentThread->Yield(); //! 这个地放做强制线程切换，测试链表插
            p->prev->next = _new;
            p->prev = _new;
            return;
        }
    }
    if (sortKey >= p->key) { // 插入最后一个
        _new->prev = p;
        p->next = _new;
        last = _new;
    }
};

// 删除第一个符合条件的节点
void *DLList::SortedRemove(int sortKey)
{
    DLLElement *p = first;
    if (p == NULL)
        return NULL;
    if (first->key == sortKey) {
        first = first->next;
        return first;
    };
    while (p->next != first) {
        p = p->next;
        if (p->key == sortKey) {
            p->prev->next = p->next;
```

```
            if (p->next != NULL)
                p->next->prev = p->prev;
            else
                last = p->prev;
            return p;
        }
    }
    return NULL;
};
```

- dllist-driver.cc

C++

```
//在list随机有序的插入n个随机数  两个强制转换的位置不同
void genItem2List(DLList *list, int n)
{
    int *item, key;

    // generating new rand() seed for each iteration
    static int random = 0;
    random++;
    srand(unsigned(time(0)) + random);

    for (int i = 0; i < n; i++) {
        item = new int;
        *item = rand() % NUM_RANGE;
        key = rand() % NUM_RANGE;
        cout << "Insert: key->" << key << endl;
        list->SortedInsert((void *)item, key); //差别在这
        list->ShowList1();
    }
}
void genItem2List2(DLList *list, int n,int which)
{
    int *item, key;

    // generating new rand() seed for each iteration
    static int random = 0;
    random++;
    srand(unsigned(time(0)) + random);

    for (int i = 0; i < n; i++) {
```

```
        item = new int;
        *item = rand() % NUM_RANGE;
        key = rand() % NUM_RANGE;
        printf("threadNo{%d} Insert: key->%d\n", which, key);
        list->SortedInsert2((void *)item, key); //差别在这
        list->ShowList1();
    }
}

//在list随机删除n个数
void delItemFromList(DLList *list, int n)
{
    int k, key;
    srand((int)time(0));

    for (int i = 0; i < n; i++) {
        int number = 0;
        number = list->CountList();
        if (list->IsEmpty()) {
            k = rand() % number;
            key = list->FoundKey(k);
            list->SortedRemove(key);
            cout << "Remove: key->" << key << endl;
            list->ShowList2();
        }
        else {
            cout << "List emptied." << endl;
            return;
        }
    }
}
```

- 对 `main.cc` 的修改【读取参数】

参数标记	变量名	含义
-q	`int testnum`	测试编号，用户进入不同的测试分支
-t	`int`	需要创建的

	`threadnum`	并行线程数量
-n	`int oprNum`	链表操作的元素个数

这里当不设置参数时，默认 `testnum = 1 , oprNum = 2 , threadNum = 2;`

```C++
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':
         oprNum = atoi(argv[1]);
         argCount++;
            break;
        default: // 默认
            testnum = 1;
            oprNum = 2 ;
            threadNum = 2;
            break;

    }
}
```

3. 并发程序设计

- 并发错误类型

错误类型	现象
共享内存	并行执行时一个线程可能删除 / 修改其余线程插入的元素
覆盖	并行的线程在链表同一个地方插入元素，导致其中一个被覆盖

删除	并行的线程准备删除链表中同一个元素，导致段错误
段错误	并行的线程一边删除一边插入，导致插入线程出现非法访问
断链	并行的线程在同一个地方插入元素，导致元素指针发生不一致
乱序	并行的线程在同一个地方插入元素，导致元素位置颠倒，键值大的

- 并行线程创造

```C++
//调用thread->fork生成并行线程
void toDllistTest(VoidFunctionPtr func)
{
    DEBUG('t', "Entering  toDllistTest\n");
    Thread *t;
    for (int i = 0; i < threadNum; i++)
    {
        t = new Thread(getName(i + 1));
        t->Fork(func, i + 1);
    }
}
```

4. 模拟并发错误产生并分析

- 共享内存

```C++
void DllistTest(int witch)
{ //展示双向链表
    genItem2List(list, 5);
    list->ReverseShowList();//逆序打印
    delItemFromList(list, 3);
    list->ReverseShowList();
}
```

可以描述为线程 A 向链表插入数据 -> 线程 B 向链表插入数据 -> 线程 A 从链表删除数据 -> 线程 B 从链表删除数据。这种情况下两个线程均有可能修改到对方的元素。

使用命令 `./nachos -q 2` 进入 `driverTest()` 的测试分支，截图如图：

```
[cs204542@mcore threads]$ ./nachos -q 2
Hello Nachos! I'm 22920202204542
Entering test 2
Inserting items in thread 1
Insert: key->121
插入后打印：121
Insert: key->146
插入后打印：121 146
Inserting items in thread 2
Insert: key->124
插入后打印：121 124 146
Insert: key->113
插入后打印：113 121 124 146
Removing items in thread 1
Remove: key->121
删除后打印：113 124 146
Remove: key->113
删除后打印：124 146
Removing items in thread 2
Remove: key->146
删除后打印：124
Remove: key->124
NULL
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

从图上可以看出，线程 1 插入了元素 121、146，线程 2 插入了元素 124、113，线程 1 删除了 121、113，线程 2 插入了元素 146、124。

故可知，线程 1 删除了线程 2 插入的元素 113，线程 2 删除了线程 1 插入的元素 146，这是因为删除元素是随机的。

- 覆盖

```C++
void DllistTest2(int which)
{ //覆盖和断链
  genItem2List2(list, oprNum);
  genItem2List2(list, oprNum);
}
```

与 Q1 不同的是，Q1 是在整个插入操作结束后切换线程，而 Q2 是在插入的过程中，即指针建立时进行线程切换。

```C++
void DLList::SortedInsert2(void *item, int sortKey)
{ //覆盖和断链测试
    DLLElement *p = first;
```

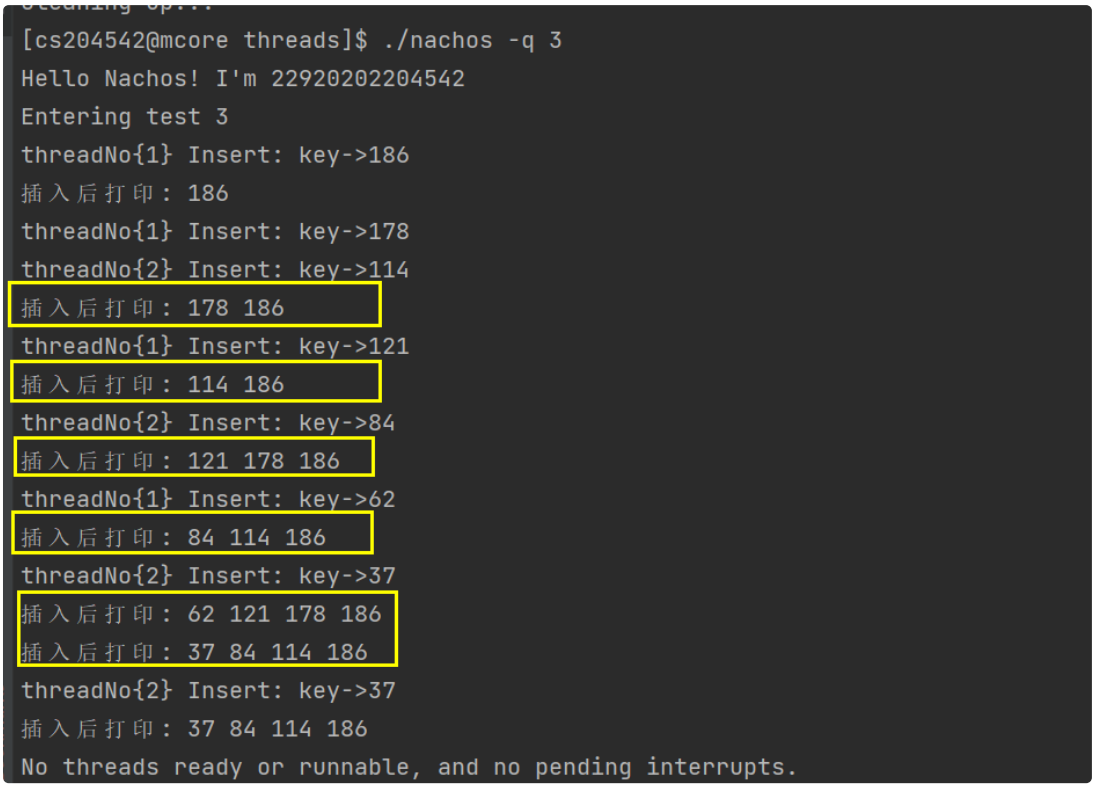
```
    DLLElement *_new = new DLLElement(item, sortKey);
    if (first == NULL)
    {
        first = _new;
        last = _new;
        return;
    }
    if (sortKey < first->key)
    { // 插入第一个
        _new->next = first;
        currentThread->Yield();
        first->prev = _new;
        first = _new;
        return;
    }
    while (p->next != NULL)
    { // 插入中间
        p = p->next;
        if (sortKey < p->key)
        {
            _new->next = p;
            currentThread->Yield();
            _new->prev = p->prev;
            p->prev->next = _new;
            p->prev = _new;
            return;
        }
    }
    if (sortKey >= p->key)
    { // 插入最后一个
        _new->prev = p;
        currentThread->Yield();
        p->next = _new;
        last = _new;
    }
};
```

```
[cs204542@mcore threads]$ ./nachos -q 3
Hello Nachos! I'm 22920202204542
Entering test 3
threadNo{1} Insert: key->84
插入后打印：84
threadNo{1} Insert: key->168
threadNo{2} Insert: key->60
插入后打印：84 168
```

线程 1、2 各插入了 4 各元素，但是最终链表长度为 6，有两个元素插入时被另一个线程的插入操作覆盖了。

- 断链

由于切换线程的位置时随机的，故 Q3 的实现与 Q2 实现的原理和代码时相同的



这里可以看出，线程 1 和线程 2 各形成了两条不同的链表。

线程 1：62→121→178→186

线程 2：37→84→114→186

两条链表在开始的时候断开了，导致了形成两条不同的链表，但是两条链表的尾部是相同的。

- 乱序

```C++
void DllistTest3(int which)
{ //乱序
    printf("Thread {%d} start.\n", which);
    if (which == 1)
    {
        InsertItem(which, list, 1);
        InsertItem(which, list, 10);
        InsertItem(which, list, 0);
        InsertItem(which, list, 5);
        PrintList(which, list); // 此处发生线程切换
        InsertItem(which, list, 3);
        InsertItem(which, list, 7);
        PrintList(which, list);
    }
    else
    { // which == 2
        PrintList(which, list);
        InsertItem(which, list, 3);
        PrintList(which, list);
    }
}
```

```
[cs204542@mcore threads]$ ./nachos -q 4
Hello Nachos! I'm 22920202204542
Entering test 4
Thread {1} start.
线程{1} 插入元素 1
线程{1} 插入元素 10
线程{1} 插入元素 0
线程{1} 插入元素 5
Thread {2} start.
打印线程{2}的list: 0 1 10
线程{2} 插入元素 3
打印线程{1}的list: 0 1 5 10
线程{1} 插入元素 3
打印线程{2}的list: 0 1 5 3 10
线程{1} 插入元素 7
打印线程{1}的list: 0 1 3 5 3 7 10
No threads ready or runnable, and no pending interrupts
```

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

在线程 2 插入元素 3 的时候，此时线程 1 插入元素 5 的过程还未结束，导致此时链表为 0→1→10，而元素 3 检测到插入的位置应该在 1 后面，但是此时线程 1 中元素 5 的插入结束，倒是元素被插入到了元素 5 的后面，导致了乱序。

问题与解决方案

1. 数据结构的实现：借鉴了一些优秀的代码，在适配本题的基础上进行了修改。
2. 链表的打印混乱：在删除和插入打印链表容易使结果十分混乱，所以在两种不同的打印前加上明显的标识。

总结

1. 通过本次实验，首先是更加熟练地使用 C++ 进行数据结构的实现和各种实验的设计。
2. 通过本次实验，深入理解几种并发程序可能导致的错误，对操作系统进程与线程的认识更加透彻。
3. 熟悉编写 Makefile 的语言规则。Makefile 可以将编译众多文件的命令汇集到一个文件了，使用 make 命令执行 Makefile ，使用编译运行程序的效率提高。
4. 熟悉了线程切换的原理和实现。`currentThread->Yield()`这个函数实际上就是将当前线程暂停，去实现另一个线程，当另一个线程完成操作之后，原线程再继续执行后续的操作，