



操作系统实验报告

| | |
|-----|--|
| 实验: | Lab1 The Trouble with Concurrent Programming |
| 专业: | 计算机科学与技术 |
| 班级: | 1 班 |
| 姓名: | 姚怀聿 |
| 学号: | 22920202204632 |

2023 年 3 月 24 日

目 录

| | | |
|----|----------------------|----|
| 一、 | 实验目的..... | 3 |
| 二、 | 实验要求..... | 3 |
| 三、 | 实验设计及关键代码实现..... | 3 |
| 1. | 多线程执行可能出现的问题..... | 3 |
| 2. | 正确编译 Nachos..... | 4 |
| 3. | 代码涉及的参数..... | 5 |
| 4. | 实现双向链表..... | 5 |
| 5. | 线程并发以及并发可能引起的问题..... | 11 |
| | 问题 1：内存共享..... | 12 |
| | 问题 2：覆盖..... | 13 |
| | 问题 3：乱序..... | 15 |
| 四、 | 遇到的问题..... | 18 |
| 五、 | 实验总结..... | 18 |

一、实验目的

1. 熟悉 Nachos 系统。了解 Nachos 的目录结构,最主要的部分是 Nachos 的 code 部分。
2. 初步了解 Makefile 的构成和相互关系。
3. 修改 Nachos 线程管理部分源代码体会多线程并发会导致的问题。

二、实验要求

1. 安装 、编译 nachos
2. 实现双向有序链表

撰写 dlist.cc、dlist.h、dlist-driver.cc 文件。文件中要包括类的定义和实现,以及有序插入 N 个随机数函数和删除函数的实现,同时打印出删除的项目。

3. 体验 nachos 线程系统

需要做的更改有:

- 1) 将 dlist.cc、dlist.h、dlist-driver.cc、dlist-driver.h 等文件拷贝到 nachos-3.4/code/threads/目录下。
- 2) 修改 Makefile.common 中的 THREAD_H、THREAD_C、THREAD_O 以保证新的文件可以被正确编译。
- 3) 根据实验内容,修改 main.cc, threadtest.cc 等文件,通过操作双向链表展现多线程并发所引发的问题。

三、实验设计及关键代码实现

1. 多线程执行可能出现的问题

1) 线程安全问题

- **原子性:** 即一个操作或多个操作,要么全部执行并且执行过程中不会被任何的因素打断,要么就都不执行
- **原子操作:** 即不会被线程调度机制打断的操作,没有上下文切换

在并发编程中很多的操作都不是原子操作,如:

操作 1: $i = 0$; 对基本数据类型变量的赋值操作是原子操作

操作 2: $i++$; 包含 3 个操作, 读取 i 的值, 将 i 加 1, 将值赋给 i

操作 3: $i = j$; 包含 2 个操作, 读取 j 的值, 将 j 的值赋给 i

操作 4: $i = i + 1$; 包含 3 个操作, 读取 i 的值, 将 i 加 1, 将值赋给 i

所以非原子操作的每个操作都可能被线程调度机制打断, 引发问题

- **可见性:** 指当多个线程访问同一个变量时, 一个线程修改了这个变量的值, 其他线程能够立即得到这个修改的值。每个线程都有自己的工作内存, 工作内存和主存间要通过 `store` 和 `load` 进行交互。假如有两个线程: 线程 1 在自己的工作内存中完成赋值操作, 却没有及时将新值刷新到主内存中。而线程 2 会从主内存中读取 i 的值, 然后加载到自己的工作内存中, 但赋值工作还没有完成, 就进行了读取。这就是可见性问题。

2) 活跃性问题: 可分为死锁、活锁和饥饿。

3) 性能问题

- 在创建或撤销进程时, 系统都要为之分配或回收进程控制块 `PCB` 及其他资源。操作系统为此所付出的开销, 明显大于创建或撤销线程时的开销。
- 在进程切换时涉及进程上下文的切换, 而线程切换时只需保存和设置少量寄存器内容, 开销很小。

本次实验主要涉及**线程安全问题**。

2. 正确编译 Nachos

`Makefile.common` 文件定义了编译链接生成一个完整的 Nachos 可执行文件所需的所有规则。我们需要把要添加的 `.h` 和 `.cc` 文件放到 `_H`、`_C` 还有 `_O` 的列表中。将新编写的 `dlist.cc`、`dlist.h`、`dlist-driver.cc` 文件加入列表中, 使 `make` 可以正常编译链接出目标文件。

```
THREAD_H = ../threads/copyright.h\  
> ../threads/list.h\  
> ../threads/scheduler.h\  
> ../threads/synch.h \  
> ../threads/synclist.h\  
> ../threads/system.h\  
> ../threads/thread.h\  
> ../threads/utility.h\  
> ../machine/interrupt.h\  
> ../machine/sysdep.h\  
> ../machine/stats.h\  
> ../machine/timer.h\  
> ../threads/dlist.h
```

```
THREAD_C = ../threads/main.cc\  
> ../threads/list.cc\  
> ../threads/scheduler.cc\  
> ../threads/synch.cc \  
> ../threads/synclist.cc\  
> ../threads/system.cc\  
> ../threads/thread.cc\  
> ../threads/utility.cc\  
> ../threads/threadtest.cc\  
> ../machine/interrupt.cc\  
> ../machine/sysdep.cc\  
> ../machine/stats.cc\  
> ../machine/timer.cc\  
> ../threads/dlist.cc\  
> ../threads/dlist-driver.cc
```

```
THREAD_0 =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 dlist.o dlist-driver.o
```

3. 代码涉及的参数

表 1 可传入的参数

| 参数标记 | 对应变量名 | 参数含义 |
|------|-----------------|-----------------------------------|
| -q | int testnum | 测试编号，用于进入不同的测试分支(默认为 1) |
| -t | int threadnum | 需要创建的并行线程数量(默认为 2) |
| -n | int oprnum | 链表操作的元素个数(默认为 2) |
| -y | bool yield_flag | 标志是否能进行线程切换(1 表示可以，0 表示不可以，默认为 0) |

表 2 不同 testnum 的不同功能

| testnum | 功能 | 指令 |
|---------|----------|---------------------|
| 1 | 测试双向链表 | -q 1 -t 1 |
| 2 | 复现进程共享问题 | -q 1 -t 2 -n 5 -y 1 |
| 3 | 复现覆盖问题 | -q 2 -t 2 -n 5 -y 1 |
| 4 | 复现乱序问题 | -q 3 -y 1 |

4. 实现双向链表

- 1) 按照 Nachos 实验指导的 3.1 部分将双向链表的定义写入 dlist.h 文件中，并作适当的添加和修改:

DLElement 类:

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

};

```

DLList 类:

```

class DLList {
    public:

        DLList(); // 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 no elements

        // routines to put/get items on/off list in order (sorted by key)

        void SortedInsert(void *item, int sortKey);

        void SortedInsert2(void *item, int sortKey);

        void *SortedRemove(int sortKey); // remove first item with
key==sortKey

        // return NULL if no such item exists

        void ShowList(int type);

        DLLElement * getFirst() { return first; }

        void setFirst(DLLElement *p) { first = p; }

    private:

        DLLElement *first; // head of the list, NULL if empty

        DLLElement *last; // last element of the list, NULL if empty

```

```
};
```

2) 在 dlist.cc 文件中根据定义写出具体的函数实现（这里只展示重要代码）

SortedInsert 函数：

```
void DList::SortedInsert(void *item, int sortKey) {  
    DLLElement *newnode = new DLLElement(item, sortKey);  
    DLLElement *ptr;  
    if(IsEmpty()) { // if is empty, newone is the only one  
        first = newnode;  
        last = newnode;  
    } else if(sortKey < first->key) {  
        newnode->next = first;  
        if(yield_flag && (testnum == 2 || testnum == 3))  
            currentThread->Yield();  
        first->prev = newnode;  
        first = newnode;  
        return ;  
    } else {  
        for(ptr = first; ptr->next != NULL; ptr = ptr->next) {  
            if(sortKey < ptr->next->key) {  
                newnode->next = ptr->next;  
                newnode->prev = ptr;  
                if(yield_flag && (testnum == 2 || testnum == 3))  
                    currentThread->Yield();  
                ptr->next->prev = newnode;  
                ptr->next = newnode;  
                return ;  
            }  
        }  
    }  
}
```

```

        // insert to the tail

        newnode->prev = last;

        if(yield_flag  &&  (testnum  ==  2  ||      testnum  ==  3))
currentThread->Yield();

        last->next = newnode;

        last = newnode;

    }

}

```

SortedRemove 函数:

```

void *DLList::SortedRemove(int sortKey) { // find the first elem that the key is
equal to sortKey and remove it

    if(IsEmpty()) return NULL;

    DLLElement *ptr;

    void *TB_return;

    if(first->key == sortKey) { // if the first is equal to the sortKey, then delete
it

        first = NULL;

        last = NULL;

    } else {

        for(ptr = first->next; ptr->next != NULL; ptr = ptr->next) {

            TB_return = ptr->item;

            if(ptr->key == sortKey) {

                ptr->prev->next = ptr->next;

                ptr->next->prev = ptr->prev;

                delete ptr;

            }

        }

        if(ptr->key == sortKey) {

```



```

        ptr->prev->next = NULL;

        last = ptr->prev;

        TB_return = ptr->item;

        delete ptr;

    } else return NULL;

}

return TB_return;

}

```

- 3) 在 `dllist-driver.cc` 中实现两个函数，其中一个函数需要插入 N 个元素，另外一个函数需要从双向链表的头部删除 N 个元素并将它们打印到终端。这两个函数的参数都是一个 `int` 整形 N 和一个指向双向链表的指针。

`genItem2List` 函数：

```

void genItem2List(int n, DLList *dllist) { // generate n random keys and the
dllist points to the list

    int *item, key;

    if(!seed) {

        srand(unsigned(time(0)));

        seed = 1;

    }

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

        item = new int;

        *item = rand();

        key = rand() % NUM_RANGE;

        printf("Insert %d into the list\n", key);

        dllist->SortedInsert((void *)item, key);

        printf("Insert %d into the list complete\n", key);

    }

    dllist->ShowList1();
}

```

```
}
```

delItemFromList 函数:

```
void delItemFromList(int n, DLList *dllist) { // removes N items starting from
the head of the list

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

        if(!dllist->IsEmpty()) {

            printf("Delete from the head\n");

            int keyval;

            dllist->Remove(&keyval);

            printf("Delete %d from the list\n", keyval);

        } else {

            printf("The list is empty\n");

            return ;

        }

    }

    dllist->ShowList2();

}
```

- 4) 在 threadtest.cc 中另写一个函数，调用上述两个函数，验证代码的正确性，展示双向链表。

```
void DLListTest(int which) {

    fprintf(stdout, "Insert items in thread %d\n", which);

    genItem2List(oprnum, dllist);

    if(yield_flag == true) currentThread->Yield();

    fprintf(stdout, "Remove items in thread %d\n", which);

    delItemFromList(oprnum, dllist);

}
```

5) 运行结果

在终端输入./nachos -q 1 -t 1 得到结果：

```
[cs204632@mc core threads]$ ./nachos -q 1 -t 1
Entering test 1
Insert items in thread 0
=====Inserting 60297 into the list.....=====
* 60297 *
=====Inserting 60297 into the list complete=====

=====Inserting 43674 into the list.....=====
* 43674 60297 *
=====Inserting 43674 into the list complete=====

Remove items in thread 0
=====Delete from the head.....=====
@ 60297 @
=====Delete 43674 from the list complete=====

=====Delete from the head.....=====
NULL
=====Delete 60297 from the list complete=====

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

5. 线程并发以及并发可能引起的问题

- 首先，我们需要先了解线程切换的方法，强制的线程切换使用 `currentThread->Yield()`，这里需要注意的是需要引用 `system.h` 文件。

`currentThread` 定义在 `system.cc` 中，表示当前正在运行的线程：

```
Thread *currentThread; >→→→→→ // the thread we are running now
```

调用 `Thread` 类中的 `Yield` 方法可以使得当前线程立即放弃所占用的 CPU，使得其他可以运行的线程获得所需要的 CPU 资源从而执行其他线程下的任务：

```
void Yield(); >→→→→→ // Relinquish the CPU if any-
// other thread is runnable
```

- 修改 `threads/main.cc` 和 `threads/theadtest.cc` 文件，实现可以创建 `T` 个线程（不同步），每个线程先插入 `N` 个元素，再依次移除并打印。所有线程共用一个链表。
- 设计可能发生的线程冲突问题，并探究其发生的原因。

问题 1：内存共享

并行执行时一个线程可能会修改其他线程正在进行操作的双向链表，比如一个线程还没有完成它全部应该完成的任务时调用了 `Yield` 将资源让给其他线程，其他线程就再次对这个链表进行操作，使得链表还没有被删除就再次被插入。

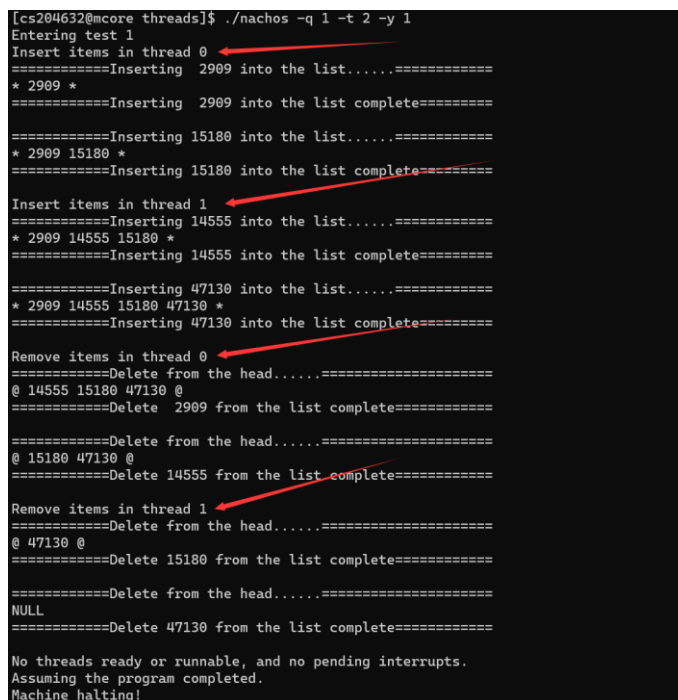
我们可以在代码中加入这样一行代码来模拟该种情况（红色为加入的代码）：

```
void DLListTest(int which) {  
  
    fprintf(stdout, "Insert items in thread %d\n", which);  
  
    genItem2List(oprnum, dllist);  
  
    if(yield_flag == true) currentThread->Yield();  
  
    fprintf(stdout, "Remove items in thread %d\n", which);  
  
    delItemFromList(oprnum, dllist);  
  
}
```

其中 `yield_flag` 是在 `threadtest.cc` 中定义的一个变量，表示是否可以强制切换线程，若值为 1 则可以，反之则不可以。通过在命令行中加入 `-y 1` 的参数可以改 `yield_flag` 的值为 1。

但这种情况比较“善良”，并不会出现报错信息，只是会删除掉别的线程所生成的随机节点。运行如下指令，可看到结果：

`./nachos -q 1 -t 2 -y 1`



```
[cs204632@mcrc threads]$ ./nachos -q 1 -t 2 -y 1  
Entering test 1  
Insert items in thread 0  
=====Inserting 2909 into the list.....=====  
* 2909 *  
=====Inserting 2909 into the list complete=====  
  
=====Inserting 15180 into the list.....=====  
* 2909 15180 *  
=====Inserting 15180 into the list complete=====  
  
Insert items in thread 1  
=====Inserting 14555 into the list.....=====  
* 2909 14555 15180 *  
=====Inserting 14555 into the list complete=====  
  
=====Inserting 47130 into the list.....=====  
* 2909 14555 15180 47130 *  
=====Inserting 47130 into the list complete=====  
  
Remove items in thread 0  
=====Delete from the head.....=====  
@ 14555 15180 47130 @  
=====Delete 2909 from the list complete=====  
  
=====Delete from the head.....=====  
@ 15180 47130 @  
=====Delete 14555 from the list complete=====  
  
Remove items in thread 1  
=====Delete from the head.....=====  
@ 47130 @  
=====Delete 15180 from the list complete=====  
  
=====Delete from the head.....=====  
NULL  
=====Delete 47130 from the list complete=====  
  
No threads ready or runnable, and no pending interrupts.  
Assuming the program completed.  
Machine halting!
```

可以看到，第一个线程先插入

然后紧接着第二个线程进行插入，最后是两个线程接连删除链表中的内容，最后链表中的内容会删空。

问题 2：覆盖

并发的线程在链表的同一个地方插入不同的元素，导致其中那个先插入的元素被覆盖。

DLListTest2 函数：

```
void DLListTest2() {  
    fprintf(stdout, "Insert items in thread %d\n", which);  
    genItem2List(oprnum, dllist);  
}
```

与问题 1 的区别在于，问题 1 是在整个插入操作执行完毕后进行线程的强制切换，而要复现出问题 2 的情况就需要在插入的过程中（即在新生成的节点与其他节点建立链接时）进行线程切换。

修改 SortInsert 函数(红色为起主要作用的代码)：

```
void DLList::SortedInsert(void *item, int sortKey) {  
    DLLElement *newnode = new DLLElement(item, sortKey);  
    DLLElement *ptr;  
    if(IsEmpty()) { // if is empty, newone is the only one  
        first = newnode;  
        last = newnode;  
    } else if(sortKey < first->key) {  
        newnode->next = first;  
        if(yield_flag && (testnum == 2 || testnum == 3))  
currentThread->Yield();  
        first->prev = newnode;  
        first = newnode;  
        return ;  
    } else {
```

```

        for(ptr = first; ptr->next != NULL; ptr = ptr->next) {
            if(sortKey < ptr->next->key) {
                newnode->next = ptr->next;
                newnode->prev = ptr;

                if(yield_flag && (testnum == 2 || testnum == 3))
currentThread->Yield();

                ptr->next->prev = newnode;
                ptr->next = newnode;
                return ;
            }
        }

        // insert to the tail
        newnode->prev = last;

        if(yield_flag && (testnum == 2 || testnum == 3))
currentThread->Yield();

        last->next = newnode;
        last = newnode;
    }
}

```

在终端执行./nachos -q 2 -t 2 -y 1 -n 5 表示执行 DLListTest2()函数并开启两个线程同时往双向链表中插入 5 个随机数，设置线程可切换，运行结果如下：

```

[cs204632@mc core threads]$ ./naches -q 2 -t 2 -y 1 -n 5
Entering test 2
Insert items in thread 0
=====Inserting 54824 into the list.....=====
* 54824 *
=====Inserting 54824 into the list complete=====
=====Inserting 34812 into the list.....=====
Insert items in thread 1
=====Inserting 42071 into the list.....=====
* 34812 54824 *
=====Inserting 34812 into the list complete=====
=====Inserting 30361 into the list.....=====
* 42071 54824 *
=====Inserting 42071 into the list complete=====
=====Inserting 44721 into the list.....=====
+ 30361 34812 54824 *
=====Inserting 30361 into the list complete=====
=====Inserting 49258 into the list.....=====
+ 30361 34812 54824 *
=====Inserting 44721 into the list complete=====
=====Inserting 49973 into the list.....=====
* 30361 34812 49258 54824 *
=====Inserting 49258 into the list complete=====
=====Inserting 23817 into the list.....=====
* 30361 34812 49973 54824 *
=====Inserting 49973 into the list complete=====
=====Inserting 6783 into the list.....=====
* 23817 30361 34812 49973 54824 *
=====Inserting 23817 into the list complete=====
* 6783 30361 34812 49973 54824 *
=====Inserting 6783 into the list complete=====
=====Inserting 64658 into the list.....=====
* 6783 30361 34812 49973 54824 64658 *
=====Inserting 64658 into the list complete=====
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

```

可以看到，两个线程共往链表中插入了 10 个元素，但是最终打印出来的结果只有 6 个，从而复现了元素覆盖的情况。

问题 3：乱序

两个线程在同一位置插入元素，可能导致顺序颠倒。

DllistTest3 函数：

```

void DllistTest3(int which) { // out of order

    fprintf(stdout, "In thread %d\n", which);

    if(which == 0) {

        InsertItem(which, dllist, 1);

        InsertItem(which, dllist, 10); // Here to switch thread

        InsertItem(which, dllist, 0);

        InsertItem(which, dllist, 5);

        PrintList(which, dllist);

        InsertItem(which, dllist, 3);

        InsertItem(which, dllist, 7);
    }
}

```

```

        PrintList(which, dllist);
    } else { // which == 1
        PrintList(which, dllist);
        InsertItem(which, dllist, 3);
        PrintList(which, dllist);
    }
}

```

InsertItem 函数:

```

void InsertItem(int which, DLLList *dllist, int keyv) {
    fprintf(stdout, "Thread %d : Insert %d to the dllist\n", which, keyv);
    dllist->SortedInsert(NULL, keyv);
}

```

为了能够保证乱序情况的出现，DLLListTest3 中的代码使用了固定参数的节点，同样线程切换的位置发生了变化，即新插入的节点的前驱指针和后继指针都已连接完毕，而指向新插入的节点的指针还未连接时进行线程转换。

运行结果:

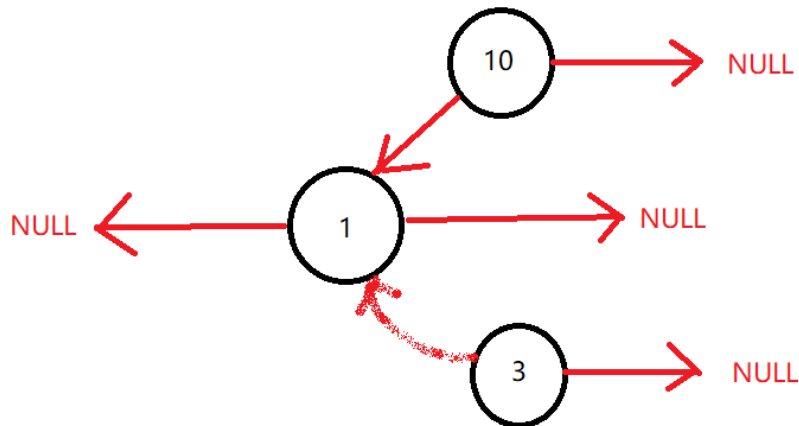
在终端输入./nachos -q 3 -y 1 可以看到

```

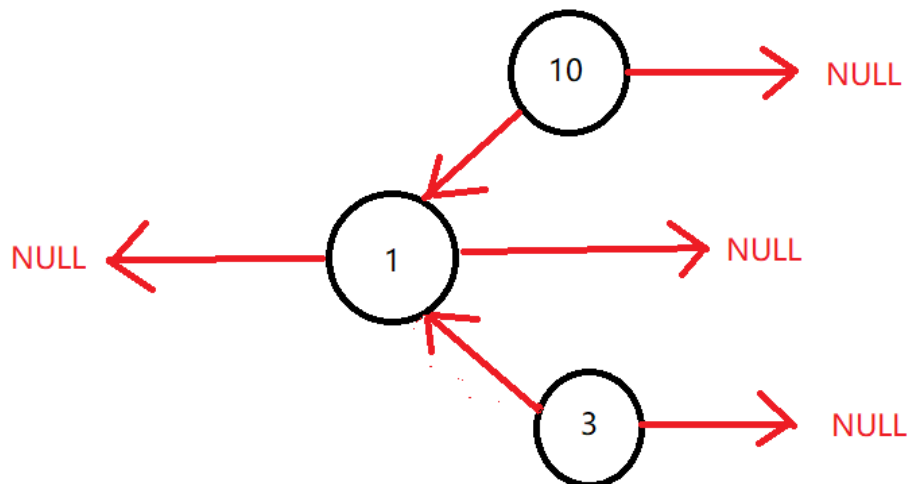
[cs204632@mcore threads]$ ./nachos -q 3 -y 1
Entering test 3
In thread 0
Thread 0 : Insert 1 to the dllist
Thread 0 : Insert 10 to the dllist
In thread 1
Print dllist in thread 1
# 1 #
Thread 1 : Insert 3 to the dllist
Thread 0 : Insert 0 to the dllist
Print dllist in thread 1
# 1 10 3 #
Thread 0 : Insert 5 to the dllist
Print dllist in thread 0
# 0 1 5 10 3 #
Thread 0 : Insert 3 to the dllist
Thread 0 : Insert 7 to the dllist
Print dllist in thread 0
# 0 1 3 5 7 10 3 #
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

```

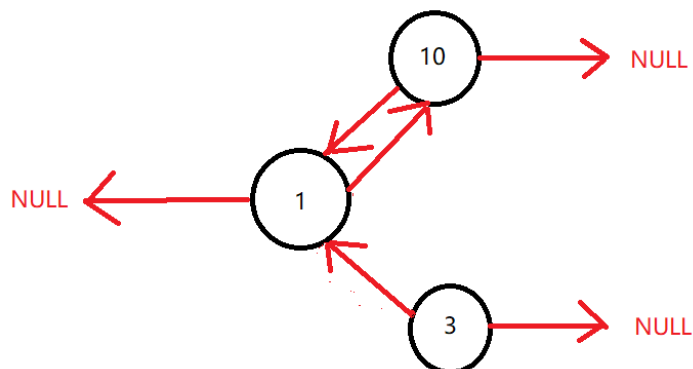

- 线程 1 想将元素 3 插入双向链表时，线程 0 还未将元素 10 完全插入双向链表，只是将元素 10 的前驱指针和后继指针都指向了正确的地方，但是还没有指针指向元素 10。



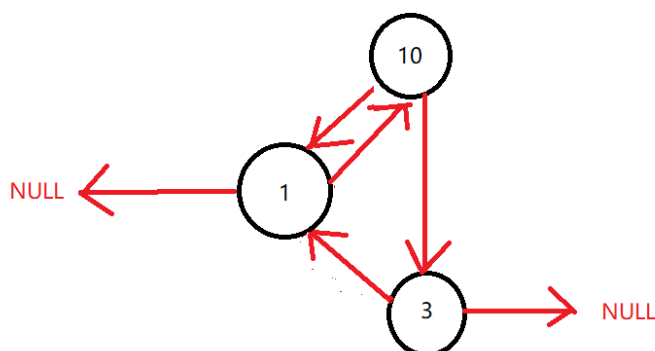
- 线程 1 插入元素 3 时发现元素 3 应该插在元素 1 的后面，并且和元素 10 一样，插了一半就换到了线程 0。



- 换回线程 0 后，线程 0 继续完成未做完的工作（将该指向元素 10 的指针修改正确），于是把元素 1 的后继指向 10。



- 在线程 0 插入下一个元素插入到一半的时候，又切换回线程 1，线程 1 继续将没插完的元素 3 插入双向链表，它会将元素 10 的后继指向元素 3。于是造成了乱序的现象



四、遇到的问题

- 一开始不太理解实验要求，像无头苍蝇一样不知道该如何进行实验。
- 双向链表不熟悉，刚开始建立和将双向链表打印出来的函数写得都有问题，于是我上网查阅资料，并简单地学习了 gdb 地用法，成功找出了 BUG，顺利完成了本次实验。
- 看网上有资料说还有可能出现断链地情况，但是我一直没有办法复现出来，或者说其实已经出现了这种情况，但是我并没有发现它出现了，于是就没有在报告中体现这一部分地内容了。

五、实验总结

- 本次实验我学习到了简单使用 nachos 系统的方法，本次实验只涉及到线程，所以通过修改 threads 里的文件编写实现目标功能的函数，并运行，观察结果并分析。
- 简单了解了编写 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，十分方便。
- 对于线程方面，主要想说的就是 `currentThread->Yield()`，这个函数实际上就是让当前线程暂时放弃资源，去执行另一个线程的任务，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。
- 本次实验中的线程并发本质上还是几个线程之间的切换，因此代码的执行顺序就非常重要，对于测试情况的设置就是基于对执行顺序的考虑，测试在不同位置切换线程可能造成的后果。