

# 线程与同步

## 提交文件说明

- `BoundedBuffer-sem.h`, `BoundedBuffer-sem.cc` 是用信号量实现的, `BoundedBuffer.h`, `BoundedBuffer.cc` 是用锁和条件变量实现的.

## Mesa style 和 Hoare Style

**Hoare** 多用于教科书描述, 在A进程发出信号时候, 等待在这个信号上的一个进程B马上运行, A被暂停, A的锁给B, 当B运行完成, 轮到A运行。实际上多数操作系统实现时候采用**Mesa**, 也就是说上述的B要等A运行完后再运行。可以在 `signal` 或者 `release` 及 `V` 操作之后, 或者保证A内所有操作具有原子性 (关中断或者用锁) 再切换线程, 保证A结束后再到其他进程。

## 以信号量实现锁和条件变量

### 目标描述及三者原语间区别

信号量 (Semaphore) 和锁配合条件变量是实现线程同步与互斥的两种等价方式。

线程的三个主要同步原语: 锁, 信号量和条件变量, 本过程中要求利用信号量实现锁和条件变量。

- **信号量: 信号量强调线程 (或进程) 之间的同步:** “信号量用于多线程和多任务同步。当一个线程完成某个动作时, 它通过信号量告诉其他线程, 其他线程执行某些动作。(当每个人都在 `sem_wait`, 它们在那里阻塞。) 当信号量是单值信号量时, 它也可以完成对资源的互斥访问。
- **互斥锁 (也称为锁) 强调资源访问的互斥:** 互斥用于多线程多任务互斥中, 一个线程占用一定的资源, 然后其他线程无法访问, 直到该线程被解锁, 其他线程可以使用此资源。例如, 有时会锁定对全局变量的访问, 并且在操作完成之后, 将其解锁。有时锁和信号灯会同时使用”, 换句话说, 信号量并不一定要锁定某个资源, 而是过程中的一个概念。例如, 有两个线程A和B。B线程必须等待A线程完成特定任务, 然后才能继续以下步骤。该任务不一定要锁定某个资源, 而是要执行一些计算或数据处理。线程互斥锁是“锁定特定资源”的概念。在锁定期间, 其他线程无法对受保护的数据进行操作。在某些情况下, 两者可以互换。
- **条件变量常与互斥锁同时使用, 达到线程同步的目的:** 条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足。在发送信号时, 如果没有线程等待在该条件变量上, 那么信号将丢失; 而信号量有计数值, 每次信号量 `post` 操作都会被记录
- 总而言之, 它们之间有以下区别:
  - 互斥锁必须是谁上锁就由谁来解锁, 而信号量的 `wait` 和 `signal` 操作不必由同一个线程执行。
  - 互斥锁要么被锁住, 要么被解开, 和二值信号量类似
  - 互斥锁是为上锁而优化的; 条件变量是为等待而优化的; 信号量既可用于上锁, 也可用于等待, 因此会有更多的开销和更高的复杂性
  - 互斥锁, 条件变量都只用于同一个进程的各线程间, 而信号量 (有名信号量) 可用于不同进程间的同步。当信号量用于进程间同步时, 要求信号量建立在共享内存区。
  - 信号量有计数值, 每次信号量 `signal` 操作都会被记录, 而条件变量在发送信号时, 如果没有线程在等待该条件变量, 那么信号将丢失。

### 锁和互斥量实现过程

## 利用Semaphore 实现锁

### Lock 的定义及初始化

互斥锁可以用一个信号量实现，因此类定义中需要定义一个信号量并初始化为1，并且根据互斥锁的定义，要求对其加锁和解锁的线程必须为同一个线程，因此用owner来记录占有锁的线程，初始化时定义为NULL，以此方便判断当前线程，从而控制释放锁的有效性。

```
class Lock {
public:
    Lock(char* debugName);           // initialize lock to be FREE
    ~Lock();                         // deallocate lock
    char* getName() { return name; } // debugging assist

    void Acquire(); // these are the only operations on a lock
    void Release(); // they are both *atomic*

    bool isHeldByCurrentThread(); // true if the current thread
                                   // holds this lock. Useful for
                                   // checking in Release, and in
                                   // Condition variable ops below.

private:
    char* name; // for debugging
    // plus some other stuff you'll need to define
    Semaphore *sem;
    Thread *owner;
};
```

### 锁的类定义

```
Lock::Lock(char* debugName)
{
    name = debugName;
    owner = NULL;
    sem = new Semaphore(debugName, 1);
}

Lock::~Lock()
{
    delete sem;
}
```

### 锁的构造和析构函数

### Lock::Acquire()的函数实现

- 对锁中的信号量成员执行P操作，信号量值减1，当值变为0时，不再允许线程继续执行占有锁，需要将其加入阻塞队列，直到另一个线程释放了锁，再进行调度，这样就保证了只有一个进程占有锁，其他试图获取锁的线程都将被阻塞。

```
bool Lock::isHeldByCurrentThread()
{
    return(currentThread == owner);
}
```

判断锁是否被当前线程占据

```
void Lock::Acquire()
{
    ASSERT(!isHeldByCurrentThread()); // 这里同时考虑了锁没用被占据(NULL)
    和被其他线程占据了的情况
    sem->P();
    owner = currentThread;
}
```

Lock::Acquire实现函数

#### Lock::Release()的函数实现

- 对锁的信号量执行V操作，信号量+1，由Acquire()函数可知，信号量在锁的实现条件下只能在0和1之间，当对信号量执行V操作，唤醒一个阻塞线程，表示锁被释放，锁的拥有者为NULL，这样就回到初始状态，等待下一次再被请求。

```
void Lock::Release()
{
    ASSERT(isHeldByCurrentThread()); // 只有同一个线程才能解开自己加的锁
    sem->V();
    owner = NULL;
}
```

### 利用Semaphore实现条件变量

#### Condition的定义与初始化

利用value值来确定有多少的线程被阻塞，并用一个初始化值为0的信号量来实现条件变量，并用刚刚实现的锁heldlock来判断信号量和互斥锁是否关联。

```
class Condition {
public:
```

```

Condition(char* debugName);    // initialize condition to
                               // "no one waiting"
~Condition();                 // deallocate the condition
char* getName() { return (name); }

void Wait(Lock *conditionLock); // these are the 3 operations on
                               // condition variables; releasing the
                               // lock and going to sleep are
                               // *atomic* in Wait()
void Signal(Lock *conditionLock); // conditionLock must be held by
void Broadcast(Lock *conditionLock); // the currentThread for all of
                                   // these operations

private:
char* name;
// plus some other stuff you'll need to define
Semaphore *sem;
int value;
Lock* heldLock;
// for ensuring every call to Signal and Wait passes an associated
mutex
};

```

condition的类定义

```

Condition::Condition(char* debugName)
{
    name = debugName;
    sem = new Semaphore(debugName, 0);
    value = 0;
}

Condition::~~Condition()
{
    delete sem;
}

```

condition的构造和析构函数

### Condition::Wait()的实现

首先记录下当前的互斥锁，然后解开相应的互斥锁并执行P操作等待条件发生变化。并且任务要求条件变量遵循Mesa风格的语义。当信号或广播唤醒另一个线程时，它只是将线程放在就绪列表中，唤醒的线程负责重新获取锁（此重新获取在Wait()中进行），因此Wait()中还需实现另一个线程重新上锁的操作。

```

void Condition::Wait(Lock* conditionLock)
{
    heldLock = conditionLock;

```

```
        conditionLock->Release();  
        value++;  
        sem->P();  
        conditionLock->Acquire();  
    }
```

### Condition::Signal()的实现

先验证当前的环境变量和互斥锁相关联，如果存在阻塞线程，则执行信号量的V操作，唤醒该线程。

```
void Condition::Signal(Lock* conditionLock)  
{  
    ASSERT(conditionLock == heldLock || heldLock == NULL);  
    // ensure every call to Signal and Wait passes an associated mutex  
  
    if(value>0) {  
        sem->V();  
        value--;  
    }  
}
```

### Condition::Broadcast()的实现

与Signal()函数类似，只是通过循环唤醒所有被wait函数阻塞在某个条件变量上的线程直到阻塞线程线程数value为空。

```
void Condition::Broadcast(Lock* conditionLock)  
{  
    ASSERT(conditionLock == heldLock || heldLock == NULL);  
    // ensure every call to Broadcast and Wait passes an associated mutex  
  
    while(value>0) {  
        sem->V();  
        value--;  
    }  
}
```

## 利用锁和条件变量修改并测试双向链表

之前的实验一实现了双向链表，并利用强制线程的切换发现了一些由于不互斥导致的错误，由此实验二在实验一的基础上需要实现线程间的互斥，现在根据实现了的锁和条件变量，编写`synchdllist.cc`和`synchdllist.h`实现互斥同步的双向链表。

### 同步双向链表的定义和初始化

保证线程安全只需要定义一个锁和判断链表是否为空的条件变量，测试只调用了`dllist-driver.cc`中的两个函数

```
class SynchDLList {
public:
    SynchDLList();           // initialize a synchronized dllist
    ~SynchDLList();          // de-allocate a synchronized dllist

    void *Remove(int *keyPtr);    // remove the first item from the
    // the dllist, waiting if
    // the dllist is empty
    // apply function to
    // every item in the dllist

    void SortedInsert(void *item, int sortKey); // routines to put/get
    // items on/off
    // dllist in order (sorted by key)

private:
    DLList *list;             // the unsynchronized dllist
    Lock *lock;               // enforce mutual exclusive access to the
    // dllist
    Condition *listEmpty;     // wait in Remove if the dllist is empty
};
```

同步双向链表的类定义

```
SynchDLList::SynchDLList()
{
    list = new DLList();
    lock = new Lock("list lock");
    listEmpty = new Condition("list empty cond");
}
SynchDLList::~SynchDLList()
{
    delete list;
    delete lock;
    delete listEmpty;
}
```

SynchDLList类的构造和析构函数

***\*SynchDLList::Remove(int keyPtr)的实现***

- 实现删除链表的头节点操作，实现线程安全的链表头部结点删除：

- 一是保证链表互斥访问，为此需要加锁；
- 不能操作空链表，为此需要通过条件变量阻塞等待链表非空。

```
void * SynchDLList::Remove(int *keyPtr)
{
    void *item;
    lock->Acquire();           // enforce mutual exclusion
    while (list->IsEmpty()==false)
        listEmpty->Wait(lock); // wait until list isn't empty
    item = list->Remove(keyPtr);
    lock->Release();
    return item;
}
```

SynchDLList::Remove()函数的实现

#### **\*SynchDLList::SortedInsert(void item, int sortKey)的实现**

- 实现链表的结点插入，需要实现链表的访问互斥，加锁即可，另外，成功结点之后链表必定为空，此时可以唤醒一个被条件变量阻塞的线程。

```
void SynchDLList::SortedInsert(void *item, int sortKey)
{
    lock->Acquire();           // enforce mutual exclusive access to
    the list
    list->SortedInsert(item,sortKey);
    listEmpty->Signal(lock);    // wake up a waiter, if any
    lock->Release();
}
```

- 相应dllist-driver.cc和 threadtest.cc 也要进行修改，调用链表的测试函数。

dllist-driiver.cc中除了需要将出现的DLList类修改为SynchDLList类之外，dllist-driver.cc中的RemoveItems()函数不再需要判断链表非空(加上也可以)，依赖头文件也需要改成synchdllist.h

```
void GenerateItems(SynchDLList *list,int N,int which)
{
    int tmpKey;
    for (int i = 0; i < N; ++i) {
        //tmpKey = rand() % 100;
        tmpKey = getRandom();
        list->SortedInsert(NULL, tmpKey);
        printf("\nThread %d Insert: %d\n", which, tmpKey);
        // list->Print();
    }
}
```

```

}
void RemoveItems(SynchDLLList *list,int N,int which)
{
    int tmpKey;
    for(int i=0;i<N;i++) {
        list->Remove(&tmpKey);
        printf("\nThread:%d Remove:%d\n",which,tmpKey);
        // list->Print();
    }
}

```

dllist-driver.cc的改动

threadtest.cc中需要写一个单独的函数来调用双向链表的值，并且使用main调用命令行参数来设定参数。

```

void DLListThread(int which)
{
    GenerateItems(list, itemNum, which);
    RemoveItems(list, itemNum, which);
}
void ThreadTest1()
{
    list = new SynchDLLList();
    // fork thread
    for (int i = 1; i < threadNum; ++i)
    {
        Thread *t = new Thread("forked thread");
        t->Fork(DLListThread, i);
    }
    DLListThread(0);
}
void ThreadTest()
{
    switch (testnum)
    {
        case 1:
            ThreadTest1();
            break;
        case 2:
            ThreadTest2();
            break;
        case 3:
            ThreadTest3();
            break;
        default:
            printf("No test specified.\n");
            break;
    }
}

```



设调用第1个为测试双向链表

实验结果:

```
[cs182204393@mc core threads]$ ./nachos -T 3 -N 4 -t 1
itemNum:4,threadNum:3
Random 1812 is generated of Thread 0.
[1/4]Insert key 1812 to the DLList of Thread 0.
Random 1976 is generated of Thread 0.
[2/4]Insert key 1976 to the DLList of Thread 0.
Random 1927 is generated of Thread 0.
[3/4]Insert key 1927 to the DLList of Thread 0.
Random 908 is generated of Thread 0.
[4/4]Insert key 908 to the DLList of Thread 0.
[1]Removed key 908 (Head of the DLList) of Thread 0.
[2]Removed key 1812 (Head of the DLList) of Thread 0.
[3]Removed key 1927 (Head of the DLList) of Thread 0.
[4]Removed key 1976 (Head of the DLList) of Thread 0.
Random 509 is generated of Thread 1.
[1/4]Insert key 509 to the DLList of Thread 1.
Random 206 is generated of Thread 1.
[2/4]Insert key 206 to the DLList of Thread 1.
Random 1992 is generated of Thread 1.
[3/4]Insert key 1992 to the DLList of Thread 1.
Random 749 is generated of Thread 1.
[4/4]Insert key 749 to the DLList of Thread 1.
[1]Removed key 206 (Head of the DLList) of Thread 1.
[2]Removed key 509 (Head of the DLList) of Thread 1.
[3]Removed key 749 (Head of the DLList) of Thread 1.
[4]Removed key 1992 (Head of the DLList) of Thread 1.
Random 1250 is generated of Thread 2.
[1/4]Insert key 1250 to the DLList of Thread 2.
Random 1363 is generated of Thread 2.
[2/4]Insert key 1363 to the DLList of Thread 2.
Random 269 is generated of Thread 2.
[3/4]Insert key 269 to the DLList of Thread 2.
Random 875 is generated of Thread 2.
[4/4]Insert key 875 to the DLList of Thread 2.
[1]Removed key 269 (Head of the DLList) of Thread 2.
[2]Removed key 875 (Head of the DLList) of Thread 2.
[3]Removed key 1250 (Head of the DLList) of Thread 2.
[4]Removed key 1363 (Head of the DLList) of Thread 2.
```

## Sleep 实现锁和条件变量

和用信号量实现的最大区别是, `sleep` 没有信号量提供的原子操作性和阻塞进程队列维护, 这些都要自己写。由于开关中断和队列维护 `Semaphore` 里面已经给了很详细的例子, 只要参照着来即可。很明显, `Lock` 和 `Condition` 的队列和 `Semaphore` 必然互不相同。另外, 针对于锁, 为了实现 `isHeldByCurrentThread()`, 要一个 `Thread *owner` 保存锁的持有者。对于条件变量, 由于有队列维护, 所以信号量实现当中的 `block_num` 就可以用 `List *conditioned` 的操作例如 `IsEmpty()` 代替实

现。下面对某些关键操作解释思路和实现过程，供给其他文件调用的接口都需要操作具有原子性，实现过程如下

```
IntStatus oldLevel = interrupt -> SetLevel(IntOff);    // 关闭中断
// 程序内容
(void) interrupt -> SetLevel(oldLevel);    // 开启中断
```

- `void Lock::Acquire()`，实现获取锁。执行之前要使得当前线程不可以持有锁，可以通过 `isHeldByCurrentThread()` 判断得到结果。结果正确后，再不断等待直到锁被释放掉。锁释放时 `owner` 会变成 `NULL`，用此可以判断。在等待过程中，当前线程一律放进 `sleep()` 队列等待激活。
- `void Lock::Release()`，实现释放锁，释放锁前提是必须持有锁。判断正确后，这时候只要把 `owner` 改成 `NULL` 并选择 `sleep()` 队列中的线程运行即可。
- `isHeldByCurrentThread()` 可以用 `currentThread` 和 `owner` 比较实现。
- `void Condition::Wait(Lock *conditionLock)`，条件变量的等待操作，首先记录下当前互斥锁，然后解开相应的互斥锁并执行P操作等待条件发生变化。由于要求条件变量遵循 *Mesa* 风格的语义。当信号或广播唤醒另一个线程时，它只是将线程放在就绪列表中，唤醒的线程负责重新获取锁（此重新获取在 `Wait()` 中进行），因此 `Wait()` 中还需实现另一个线程重新上锁的操作。
- `void Condition::Signal(Lock *conditionLock)` 先验证当前的环境变量和互斥锁相关联，如果存在阻塞线程，则将阻塞线程调出阻塞队列，转为就绪态。
- `void Condition::Broadcast(Lock *conditionLock)`，一直把阻塞队列进程转换为就绪态度，直到队列为空。

下面进行测试。原来用 `currentThread -> Yield()` 来模拟线程并行过程中可能出现的执行顺序不同导致的结果不同。例如

```
Thread 1
A
B
Thread 2
C
```

使得ABC和ACB结果不同即可反映出错误。运行结果如下

ABC，没有切换线程

```
Current DLList :
key1 : -1
key2 : 12
key3 : 32
key4 : 32
key5 : 52
key6 : 54
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

ACB，线程切换了

```
Thread 0 Inserted 5 number

Current DLLlist :
key1 : 12
key2 : 32
key3 : 32
key4 : 52
key5 : 54
thread yielding ...

Thread 1 Removed an element 12

Current DLLlist :
key1 : 32
key2 : 32
key3 : 52
key4 : 54
thread yielded ...
```

最终结果

```
Current DLLlist :
key1 : 30
key2 : 31
key3 : 32
key4 : 32
key5 : 52
key6 : 54
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

下面处理方法是

```
Lock *lock = new Lock("DLLlist lock");
Thread 1
lock -> Acquire();
A
currentThread -> Yield();
B
lock -> Release()

Thread 2
lock -> Acquire()
C
lock -> Release()
```

结果如下，线程切换后由于另一个线程没有锁所以阻塞，然后又返回到原来线程执行（总共只有两个线程）。

```

Current DLLList :
key1 : 12
key2 : 32
key3 : 32
key4 : 52
key5 : 54
thread yielding ...
thread yielded ...

Thread 0 Prepend an element

```

链表内容如下

```

DLLlistoperation 1 released the lock
DLLlistoperation 2 acquired the lock

Thread 1 Removed an element -2

Current DLLList :
key1 : -1
key2 : 12
key3 : 32
key4 : 32
key5 : 52
key6 : 54
DLLlistoperation 2 released the lock
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

```

最终结果和ABC相同，错误得到避免。

## 实现一个线程安全的表结构

线程安全的表定义为一个简单的固定大小的表，由“size”个条目组成，每个条目都有一个指向对象的指针，表中每个对象都有一个索引对应其在表中的位置。

下面是给出的Table.h，要求补充完整并实现自己的Table.cc，包括所有的函数。

```

class Table {
public:
    // create a table to hold at most 'size' entries.
    Table(int sizeVal);
    // allocate a table slot for 'object'.
    // return the table index for the slot or -1 on error.
    int Alloc(void *object);
    // return the object from table index 'index' or NULL on error.
    // (assert index is in range). Leave the table entry allocated
    // and the pointer in place.
    void *Get(int index);
    // free a table slot
    void Release(int index);

```

```
private:
    // Your code here.
};
```

## 实现过程

### 线程安全表类

线程安全的表包含有表本身（`table`），表的大小（`size`），表中已经使用的空间大小（`used`），一个锁（`lock`）以及条件变量用于阻塞表满的情况（`table_full`），最后还有构造函数。

```
class Table {
public:
    Table(int sizeVal);
    ~Table();
    int size; // size of the table
    int used; // used size of the table
    int **table;
    Lock *lock;
    Condition *table_full; //table is full
};
```

### 构造函数

初始化`size`以及`used`，建立空表`table`并赋空置，新建锁`lock`以及条件变量`table_full`。

```
Table::Table(int sizeVal)
{
    size = sizeVal;
    used = 0;
    table = new int*[size];
    lock = new Lock("Table Lock");
    table_full = new Condition("Table Full");
    for(int i=0; i<size; i++)
        table[i] = NULL;
}
```

### 析构函数

删除表、锁以及条件变量。

```
Table::~~Table()
{
    delete lock;
    delete table_full;
```

```
    delete table;
}
```

## Alloc

该函数功能为给`object`分配一个表槽，返回索引，若出错则返回-1。

首先请求锁，对`used`进行判断，若等于`size`则说明当前表满，进入队列需要等待。否则，遍历找到一个`index`对应的表槽为空，填入`object`，最后释放锁，返回`index`。

```
int
Table::Alloc(void *object)
{
    int index;
    lock->Acquire();
    while(used == size) {
        printf("Table is full\n");
        table_full->Wait(lock);
    }
    for(index=0; index<size; index++) {
        if(table[index] == NULL) {
            used++;
            table[index] = (int *)object;
            break;
        }
    }
    lock->Release();
    return index;
}
```

## Get

该函数功能为返回表索引`index`中的对象，若出错返回NULL。

首先对`index`进行合法性判断，若合法，则获得锁，取出表中索引为`index`的对象，释放锁，最后返回该对象；若不合法直接返回NULL。

```
void*
Table::Get(int index)
{
    void* item = NULL;
    if(index < 0 || index >= size) {
        printf("Error Index\n");
        return item;
    }
    else {
        lock->Acquire();
        item = table[index];
    }
}
```

```
        lock->Release();  
        return item;  
    }  
}
```

## Release

该函数功能为释放某一表槽中的对象。

与`Get()`方法相似，首先对`index`进行合法性判断，若合法，则获得锁，删除表中索引为`index`的对象，并唤醒一个由于表满阻塞的线程（若有线程在等待），最后释放锁；若不合法则输出错误提示信息。

```
void  
Table::Release(int index)  
{  
    if(index < 0 || index >= size) {  
        printf("Error Index\n");  
    }  
    else {  
        lock->Acquire();  
        table[index] = NULL;  
        used--;  
        table_full->Signal(lock);  
        lock->Release();  
    }  
}
```

## 设计程序测试验证

修改`threadtest.cc`文件，使其创建两个线程在并发的情况下修改表，我们这里将线程安全表定义为全局变量。

```
Table *t1 = new Table(2);
```

## ThreadTest3

创建两个线程，一个调用`TableOperation1`，用于模拟数据生产者，另一个调用`TableOperation2`，模拟数据的消费者。

```
void ThreadTest3()  
{  
    DEBUG('t', "Entering ThreadTest3");  
    Thread *t = new Thread("forked thread");  
    t->Fork(TableOperation2, 1);  
    TableOperation1(0);  
}
```

## TableOperation1

该函数模拟生产者动作，循环三次将data存入表t1中，并打印信息。

```
void
TableOperation1(int which)
{
    char data[3][4] = {"123", "456", "abc"};
    for(int i=0; i<3; i++) {
        int index = t1->Alloc(data[i]);
        printf("Thread %d: Alloc %s to table[%d]\n", which, data[i],
index);
    }
}
```

## TableOperation2

该函数模拟消费者动作，循环两次将表t1中的内容取出，并打印信息。

```
void
TableOperation2(int which)
{
    for(int i=0; i<2; i++) {
        void *item;
        item = t1->Get(i);
        printf("Thread %d: Get %s to table[%d]\n", which, item, i);

        t1->Release(i);
        printf("Thread %d: Release table[%d]\n", which, i);
    }
}
```

## 实验结果



```
[cs182204308@mc core threads]$ ./nachos -q 3
testnum : 3
Hello Nchoos! Here We Go!
Thread 0: Alloc 123 to table[0]
Thread 0: Alloc 456 to table[1]
Table is full
Thread 1: Get 123 to table[0]
Thread 1: Release table[0]
Thread 1: Get 456 to table[1]
Thread 1: Release table[1]
Thread 0: Alloc abc to table[0]
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 210, idle 0, system 210, 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...
```

可以看到在线程0插入两个对象后，表已满，于是切换至线程1运行，取出两个对象并删除后，再次切换给线程0插入一个对象，程序结束。另外，从实验结果中也可以看出条件变量所采用的是Mesa语义，在线程1取出一个对象后，没有立刻唤醒线程0继续执行，而是等到线程1结束，线程0才继续执行。

## 实现大小受限的安全缓冲区

### 实现内容

`BoundedBuffer.h`文件中需要实现`Read`函数和`Write`函数，并且保证是其线程安全的，需要采用信号量或者锁和条件变量来实现。其中`Read`和`Write`函数的原型如下：

```
void Read(void *data, int size);
void Write(void *data, int size);
```

其中`data`为`void *`类型的指针，代表将数据指针送入函数前需要先强制转化为`void *`类型的指针，在函数内部再将其转化为`Buffer`内部数据类型对应的指针`char *`，`size`为`int`整形变量代表读出或者写入的`data`的长度，这个长度可以超过`Buffer`内部缓冲区的大小，也就是在长度大于0的情况下就不需要报错，该问题就类似于有限缓冲区生产者/消费者问题。接下来4.1和4.2分别介绍两种不同版本的实现方式。

### 信号量版本

采用信号量实现，其重要的点就是考虑实现一个大小受限并且线程安全的缓冲区应当由多少信号量来实现，很显然答案应该有三个，在`BoundedBuffer.h`中声明三个信号量分别为：

```
Semaphore *sizeofBuffer; // 该信号量用于互斥，表示缓冲区大小
Semaphore *rwem;         // 该信号量用于互斥，保证写和读缓冲区操作互斥
Semaphore *DataSize;      // 该信号量用于同步，表示当前缓冲区数据长度
```

另外，还需要有**writeIn**和**readOut**指针分别指向当前缓冲区的写入和读出的位置，因为缓冲区大小受限因此该指针是在有限的范围内不断循环，当然表示缓冲区长度和指向缓冲区存储内容的指针也必不可少，因此其类声明定义如下：

```
class BoundedBuffer {
public:
    // create a bounded buffer with a limit of 'maxsize' bytes
    BoundedBuffer(int maxsize);
    ~BoundedBuffer();

    // read 'size' bytes from the bounded buffer, storing into 'data'.
    // ('size' may be greater than 'maxsize')
    void Read(void *data, int size);

    // write 'size' bytes from 'data' into the bounded buffer.
    // ('size' may be greater than 'maxsize')
    void Write(void *data, int size);

private:
    int BufferSize;
    int writeIn;           // 写入的当前位置
    int readOut;           // 读出的当前位置
    char *Data;            // Buffer内的数据
    Semaphore *sizeofBuffer; // 该信号量用于互斥，表示缓冲区大小
    Semaphore *rwem;       // 该信号量用于互斥，保证写和读缓冲区操作互斥
    Semaphore *DataSize;   // 该信号量用于同步，表示当前缓冲区数据长度
};
```

接下来编写构造函数和析构函数，析构函数只要将Buffer的存放数据的指针释放，并且释放定义好的三个信号量即可，构造函数除了开辟一个指定**BufferSize**大小的缓冲区，初始化**writeIn**和**readOut**外，还需要初始化三个信号量。三个信号量的初始值可以通过其作用得到，**sizeofBuffer**作用用于限制写入的内容不超过缓冲区大小，那么其值就应当大于0，且等于**Buffer**的长度，**rwem**作用用于保证写和读缓冲区操作互斥，因此其应当初始化为1，而**DataSize**其在可以表示当前缓冲区可读长度的同时，主要用于同步读写操作，因此其应当初始化为0,这样就可以得到构造函数和析构函数。

```
BoundedBuffer::BoundedBuffer(int maxsize){
    BufferSize = maxsize;
    writeIn = 0;
    readOut = 0;
    Data = new char[maxsize + 1];
    sizeofBuffer = new Semaphore("sizeofBuffer Semaphore",maxsize);
    rwem = new Semaphore("rwem Semaphore",1);
    DataSize = new Semaphore("DataSize Semaphore",0);
}
BoundedBuffer::~BoundedBuffer(){
    delete Data;
```

```

delete sizeofBuffer;
delete rwem;
delete DataSize;
}

```

接下来编写读写函数，首先先将读和写两个操作作用代码的形式表示：

```

读：
temp[i] = Data[readOut];
readOut = (readOut + 1) % BufferSize;
写：
Data[writeIn] = temp[i];
writeIn = (writeIn + 1) % BufferSize;

```

由于每次调用函数都将一次性读写多个字符，因此这里应当采用循环的方式，那么为了保证每一次读写都能够受到信号量的约束，显然信号量应当定义在循环内部。首先先看写函数，在每次写之前应当先确定当前缓冲区是否还有空间可以满足写一个字符的请求，因此需要在开始调用`sizeofBuffer->P()`，接下来代表写函数拿到了可以往一个缓冲区写入一个字符的许可，那么就可以调用`rwem->P()`来进入写入临界区，防止在写入的同时另一边在读出。在写入完成后需要释放互斥信号量`rwem->V()`，并且由于写入了一个字符，缓冲区长度加一，对应的调用一次`DataSize->V()`来通知读函数当前可以读入一个字符，那么就可以实现如下：

```

void BoundedBuffer::Write(void *data, int size){
    char *temp = (char *)data;
    ASSERT(size > 0 && data != NULL);
    for(int i=0;i<size;i++){ //一共写入size个数据
        sizeofBuffer->P(); //semWait(sizeofBuffer)
        rwem->P(); //semWait(rwem)
        Data[writeIn] = temp[i];
        writeIn = (writeIn + 1) % BufferSize;
        rwem->V(); //senSignal(rwem)
        DataSize->V(); //senSignal(DataSize)
    }
}

```

同样的，读函数的思想也和写函数类似，首先调用`DataSize->P()`来确保当前缓冲区至少具有一个可以读出的字符，那么获得读出一个字符的许可后，调用`rwem->P()`来进入读入临界区，防止与写冲突，读取完成后释放互斥信号量`rwem->V()`，并且由于读出了一个字符缓冲区空出一个字符的空间可以供写函数继续写入，因此调用一次`sizeofBuffer->V()`来通知读函数当前可以继续写入，实现如下：

```

void BoundedBuffer::Read(void *data, int size){
    char *temp = (char *)data;
    ASSERT(size > 0 && data != NULL);
    for(int i=0;i<size;i++){ //一共读入size个数据
        DataSize->P(); //semWait(DataSize)

```

```

        rwem->P();           //semWait(rwem)
        temp[i] = Data[readOut];
        readOut = (readOut + 1) % BufferSize;
        rwem->V();           //senSignal(rwem)
        sizeofBuffer->V();    //senSignal(sizeofBuffer)
    }
}

```

## 条件变量版本

采用条件变量实现，思路大体与信号量相同，但是此时应当考虑如何采用锁和条件变量来代替原先三个信号量所能实现的功能。此时需要一个锁和两个条件变量，其作用分别为：

```

Lock *mutex;           // 与条件变量配合使用的锁
Condition *BuffernotFull; // 条件变量 : buffer不再为满
Condition *BuffernotEmpty; // 条件变量 : buffer不再为空

```

其余类定义和信号量相同这里不再赘述。

这里主要看其函数的实现，在构造函数和析构函数中，正常创建和删除即可：

```

BoundedBuffer::BoundedBuffer(int maxsize){
    BufferSize = maxsize;
    writeIn = 0;
    readOut = 0;
    Data = new char[maxsize + 1];
    mutex = new Lock("Buffer Lock");
    BuffernotFull = new Condition("Buffer not full cond");
    BuffernotEmpty = new Condition("Buffer not empty cond");
}
BoundedBuffer::~~BoundedBuffer(){
    delete Data;
    delete mutex;
    delete BuffernotFull;
    delete BuffernotEmpty;
}

```

接下来是读函数和写函数，要分清这里和信号量实现的区别，在信号量中，我们借助信号量本身的值来限制写入长度不超过缓冲区以及同步读和写的操作，但是在这里条件变量只有等待和唤醒为了实现之前的功能就要在条件变量外部添加一层循环，循环中的判断语句就成为了限制写入长度以及同步读出的重要一环，此时 `writeIn` 和 `readOut` 两个变量在这种方法下就显得尤为重要在读函数中，为了确保当前缓冲区有内容读出，代码采用：

```

while(writeIn == readOut){
    BuffernotEmpty->Wait(mutex);
}

```

```
}

```

意为如果`writeIn`和`readOut`指针相同，则没有可以从缓冲区读出的内容，读函数继续等待，直到`writeIn`向前推进为止。在写函数中，为了确保当前缓冲区仍然有空间可以写入，代码采用：

```
while((writeIn+ 1) % BufferSize == readOut){
    BuffernotFull->Wait(mutex);
}
```

意为如果`writeIn`的下一个位置与`readOut`重合，缓冲区已满，不可以写入，写函数继续等待，直到读函数读出至少一个字符为止。当然上面两个激活条件“直到`writeIn`向前推进为止”和“直到读函数读出至少一个字符为止”都需要对方进行通知，而这种通知由于采用的是`mesa`语义，通知唤醒后不会马上运行而是会再次检查条件，不用担心出现超过数量的读写进程同时进入临界区，此时通知应当放在读出写入之后，释放锁之前。综上所述，两个函数的实现具体如下：

```
void BoundedBuffer::Write(void *data, int size){
    char *temp = (char *)data;
    ASSERT(size > 0 && data != NULL);
    for(int i=0;i<size;i++){ //一共写入size个数据
        mutex->Acquire();
        while((writeIn+ 1) % BufferSize == readOut){
            BuffernotFull->Wait(mutex);
        }
        Data[writeIn] = temp[i];
        writeIn = (writeIn + 1) % BufferSize;
        BuffernotEmpty->Broadcast(mutex);
        mutex->Release();
    }
}

void BoundedBuffer::Read(void *data, int size){
    char *temp = (char *)data;
    ASSERT(size > 0 && data != NULL);
    for(int i=0;i<size;i++){ //一共读入size个数据
        mutex->Acquire();
        while(writeIn == readOut){
            BuffernotEmpty->Wait(mutex);
        }
        temp[i] = Data[readOut];
        readOut = (readOut + 1) % BufferSize;
        BuffernotFull->Broadcast(mutex);
        mutex->Release();
    }
    //printf("read : %s\n",data);
}
```

## 运行测试

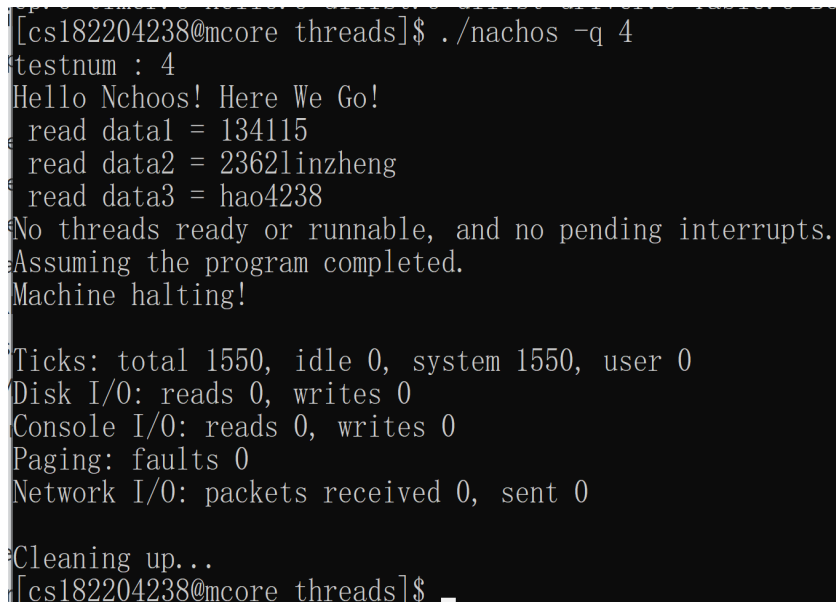
在`threadtest.cc`文件中, 定义`testnum = 4`的情况下测试线程安全的缓冲区, 我们可以创建两个线程, 一个线程专门用来读, 一个专门用来写, 观察最后读出的内容是否正确即可证明程序的正确性。两个操作函数如下:

```
void BufferOperation1(int which){
    char *data1,*data2,*data3;
    data1 = new char[10];
    data2 = new char[15];
    data3 = new char[10];
    Buffer->Read((void*)data1,6);
    Buffer->Read((void*)data2,12);
    Buffer->Read((void*)data3,7);

    printf(" read data1 = %s\n",data1);
    printf(" read data2 = %s\n",data2);
    printf(" read data3 = %s\n",data3);
}

void BufferOperation2(int which){
    Buffer->Write((void *)"1341152362",10);
    Buffer->Write((void *)"linzhenghao",11);
    Buffer->Write((void *)"4238",4);
}
```

接下来编译运行代码, 不要忘了修改`makefile`文件, 可以得到结果如下图所示:



```
[cs182204238@mcore threads]$ ./nachos -q 4
testnum : 4
Hello Nchoos! Here We Go!
 read data1 = 134115
 read data2 = 2362linzheng
 read data3 = hao4238
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1550, idle 0, system 1550, 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...
[cs182204238@mcore threads]$
```

符合预期的结果, 线程安全的缓冲区实现正确。