操作系统原理课程实验报告

> **实验 3 - NachOS 下的线程编程**
>
> 实验日期：Apr. 13th - Jun. 3rd, 2017

> **小组成员信息**
>
> OMIT

一、实验内容与要求

本实验目的在于通过使用第二次实验中实现的锁机制与条件变量的相关实现设计出更为有效的原语，进而解决较为复杂的并发问题，提高妥善处理并发编程中常见的竞争、死锁、饥饿等问题从而正确编写并发程序的能力。

本次实验的具体内容如下：

- 利用条件变量与锁机制实现 `EventBarrier` 原语，从而允许一组线程能够以同步的方式等待并相应某事件；
- 利用 NachOS 中提供的 `Timer` 类实现 `Alarm` 原语，从而允许线程可以主动调用 `Alarm::Pause` 方法休眠一定时间后由 NachOS 唤醒；
- 利用 `EventBarrier` 原语与 `Alarm` 原语解决模拟电梯运行的同步问题。

二、实验设计与代码实现

A. `EventBarrier` 原语的实现

对于 `EventBarrier` 类，我们采用如下的头文件定义：

```
class EventBarrier{
    public:
        EventBarrier();
        ~EventBarrier();
        void Wait();
        void Signal();
        void Complete();
        int Waiters();
    private:
        bool status;  // the open status of EventBarrier
        int  waitNum; // the num of waiting thread
        int  value;   // record signals
        Lock *barrierLock;        // mutex lock and conditions
        Condition *signal_con;
        Condition *complete_con;
}
```

> ***Code. 1*** EventBarrier 类的头文件定义

对于事件栅栏而言，其主要方法有 `Wait()`，`Signal()`，`Complete()`，`Waiters()` 这四个方法。线程调用 `Wait()` 时若栅栏处于打开状态，则 `Wait()` 方法直接返回，否则线程将阻塞在该栅栏的第一个条件变量上等待事件发生。当事件发生时，`Signal()` 方法将被调用，等待所有阻塞在第一个条件变量上的线程被唤醒执行 `Complete()` 。执行完 `Complete()` 的线程将阻塞在另一条件变量上，等待所有其他线程完成执行。所有线程完成执行后，`Signal()` 方法返回并打开栅栏，阻塞在另一条件变量上的变量得以释放，顺利通过栅栏。

`Waiters()` 方法返回当前阻塞在栅栏上的线程数； `barrierLock` 互斥锁则保证对共享资源操作的互斥性。

`EventBarrier` 类主要方法的具体实现如下：

```
EventBarrier::EventBarrier() {
    status = false;
    waitNum = 0;
    value = 0;
    barrierLock = new Lock("barrier Lock");
    signal_con = new Condition("singal conditon");
    complete_con = new Condition("complete condition");
}

EventBarrier::~EventBarrier() {
    delete barrierLock;
    delete signal_con;
    delete complete_con;
}

void
```

```
EventBarrier::Wait() {
    barrierLock->Acquire();
    if(status == false){
        waitNum++;
        signal_con->Wait(barrierLock);
        waitNum--;
    }
    barrierLock->Release();
}

void
EventBarrier::Signal() {
    barrierLock->Acquire(); // only one can call Signal
    status = true;
    if(Waiters()!=0) {
        // wake up other waiters
        signal_con->Broadcast(barrierLock);
        // blocked in complete condition
        complete_con->Wait(barrierLock);
    } else {
        // if no one, just continue
        status = false;
    }
    // resum the status of barrier
    barrierLock->Release();
}

void
EventBarrier::Complete() {
    barrierLock->Acquire();
    if(Waiters() == 0) {
        // the last one to release lock
        status = false;// change status
        complete_con->Broadcast(barrierLock);
    } else {
        // no the last one
        complete_con->Wait(barrierLock);
        //changeStatus_con->Wait(barrierLock);
    }
    barrierLock->Release();
}

int
EventBarrier::Waiters() {
    barrierLock->Acquire();
    int num = waitNum;
    barrierLock->Release();
```

```
    return num;
}
```

> ***Code. 2*** EventBarrier 类的主要实现

B. `Alarm` 原语的实现

对于 `Alarm` 类，我们采用如下的头文件定义：

```
class Alarm{
public:
    Alarm();
    ~Alarm();
    void Pause(int howLong);
    void awake();
    static int num; // record the num of threads in list
private:
    List  *list;
};
```

> ***Code. 3*** Alarm 类的头文件定义

闹钟原语中的主要方法只有 `Pause()` 与 `awake()` 。线程调用 `Pause()` 来使自身至少阻塞指定的 Tick 数，这可以通过在闹钟中维护一个有序列表来实现。之后，通过修改 NachOS 中的时钟中断的处理函数，我们可以在其中调用 `awake()` 方法来维护闹钟中的有序数组，并将到达唤醒时间的线程从阻塞队列移入就绪队列。

`Alarm` 类主要方法的具体实现如下：

```
int Alarm::num = 0;
Alarm::Alarm()
{ list = new List(); }

Alarm::~Alarm()
{ delete list; }

void
check(int  which) {
    // check if any threads is waiting for alarm
    // if so, switch to dummy thread
    while(Alarm::num != 0) { currentThread->Yield(); }
    DEBUG('t', "dummy thread finish\n");
    return;
}

void
Alarm::Pause(int howLong) {
    //use interup to make it atomicly
```

```
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    // calculate the waking up time
    int when = stats->totalTicks + howLong*TimerTicks;
    list->SortedInsert(currentThread, when); // insert into list
    num++;
    // create dummy thread to prevent NachOS from halting
    if (num == 1) {
        Thread *t = new Thread("dummy thread\n");
        t->Fork(check,0);
    }
    currentThread->Sleep();  // thread sleep
    (void) interrupt->SetLevel(oldLevel);   // re-enable interrupts
}

void
Alarm::awake() {
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    // traverse the list
    // and move timed-out thread to ready list
    Thread *temp;
    int when, len = num;
    for (int i=0; i < len; i++) {
        temp = (Thread*)list->SortedRemove(&when);
        if(when <= stats->totalTicks) {//time out
            scheduler->ReadyToRun(temp);
            num--;
        } else {
            // the others are postponed to now
            list->SortedInsert(temp,when);
            break;
        }
    }
    (void) interrupt->SetLevel(oldLevel);
}
```

> ***Code. 4*** Alarm 类的主要实现

对于 `system.cc` 中的时钟中断处理函数，我们做出以下修改：

```
static void
TimerInterruptHandler(int dummy) {
    if (interrupt->getStatus() != IdleMode) {
      interrupt->YieldOnReturn();
    }
    alarms->awake();
}
```

> ***Code. 5*** system.cc 中的修改

为方便引用，我们也在 `system.cc` 与 `system.h` 中加入了全局变量 `alarms` ，具体修改参见源代码。

C. 电梯问题

为模拟一座大楼、一台有限容量下的电梯问题，我们首先定义 `Building` 类与 `Elevator` 类，采用如下的头文件定义：

```
class Elevator {
   public:
     Elevator(char *debugName, int numFloors, int myID);
     ~Elevator();
     char *getName() { return name; }

     // elevator control interface: called by Elevator thread
     void OpenDoors();   // signal exiters and enterers to action
     void CloseDoors();  // after exiters are out and enterers are in
     void VisitFloor(int floor); //   go to a particular floor

     // elevator rider interface (part 1): called by rider threads.
     bool Enter();   //   get in
     void Exit();    //   get out (iff destinationFloor)
     void RequestFloor(int floor); //   tell the elevator our destination

     int  getCurrentFloor() { return currentfloor; }
     int  getOccupancy() { return occupancy; }
     int  getDirection() { return direction; }
     void changeDirection() { direction = 1 - direction; }
     void setBuilding(Building *building) { b = building; }
     bool *request; // destination for riders onboard

   private:
     char *name;
     int currentfloor;           // floor where currently stopped
     int occupancy;              // how many riders currently onboard
     int capacity;               // the capacity of elevator
     int numFloors;
     int id;
     int direction;   // 0 for going down and 1 for going up
     int status;      // 0 for working and 1 for free
     int closeDoorNum;
     EventBarrier *exit;
     Lock *con_lock;
     Condition *con_closeDoor;
     Building  *b;
};

class Floor{
```

```
public:
    EventBarrier *e;
    Floor()  { e = new EventBarrier[2]; } // 0 for down, 1 for up
    ~Floor() { delete[] e; }
};

class Building {
   public:
     Building(char *debugname, int numFloors, int numElevators);
     ~Building();
     char *getName() { return name; }
     // elevator rider interface (part 2): called by rider threads
     void CallUp(int fromFloor);// signal an elevator we want to go up
     void CallDown(int fromFloor); //   ... down
     Elevator *AwaitUp(int fromFloor); // wait for elevator
     Elevator *AwaitDown(int fromFloor);

     Floor *getFloors() { return floors; }
     bool  *getSrcUp() { return srcUp; }
     bool  *getSrcDown() { return srcDown; }
     Lock  *getLock() { return mutex; }
     void  RunElev(int eid = 0);

   private:
     char *name;
     int floorNum;
     Elevator *elevator;
     Floor *floors;
     bool  *srcUp;
     bool  *srcDown;
     Lock  *mutex;
};
```

> ***Code. 6*** 电梯问题相关类的头文件定义

对于电梯问题，我们定义了一个大楼类 `Building` 与一个电梯类 `Elevator` 进行模拟。对于大楼类，我们引入了楼层 `Floor` 类。

对于一幢 F 层楼的大楼，每层楼我们设置了上行与下行方向的两个事件栅栏存放于 `Floor` 对象中，而对于每座大楼，则均有一个由 F 个 `Floor` 对象构成的数组。大楼类对乘客提供了 `CallUp()`，`CallDown()`，`AwaitUp()`，`AwaitDown()` 四个方法，分别对应乘客按键请求上行 / 下行电梯与乘客在上行 / 下行事件栅栏处等待电梯抵达。而对于电梯，大楼提供了 `srcUp` 与 `srcDown` 两个布尔数组指示在某一楼层有无乘客请求上行 / 下行电梯。

对电梯类，在其中我们定义了以下电梯行为与关键成员变量：

- `OpenDoors()` ：开启梯门，到达目的层的乘客先出梯，之后等待在本层并与电梯运行方向相同的乘客入梯（在电梯容量允许的前提下）；
- `CloseDoors()` ：在完成乘客入梯与出梯后关闭梯门；

- `VisitFloor(int)`：前往指定楼层，实验中使用 `Alarm` 类模拟电梯移动耗时；
- `Building *b`：指向本电梯绑定到的大楼对象；
- `int capacity`：电梯最大容量；
- `EventBarrier *exit`：为等待在某层出梯的乘客设置的事件栅栏；
- `bool *request`：指示目前梯内乘客目的楼层的布尔数组。

我们也在电梯类中为乘客定义了以下方法：

- `Enter()`：尝试进入电梯，如果电梯容量已满函数将返回 False；若成功进入电梯，乘客线程将通过 `RequestFloor(int)` 设置自己的目的地楼层，并阻塞到 `exit` 中对应楼层的事件栅栏上；
- `Exit()`：到达楼层后离开电梯；
- `RequestFloor(int)`：请求电梯前往某层。

本部分关键代码实现如下：

```
Elevator::Elevator(char *debugName, int numFloors, int myID) {
    name = debugName;
    this->numFloors = numFloors;  // floor of building
    id = myID; // mark one elevator
    request = new bool[numFloors+2];
    exit = new EventBarrier[numFloors+2];//barrier for going out
    con_lock = new Lock("lock for occupancy");
    con_closeDoor = new Condition("condition for close door");
    occupancy = 0;// can setting
    capacity = 3; // setting capacity
    currentfloor = 1;
    direction = 0;
}

Elevator::~Elevator() {
    delete[] request;
    delete[] exit;
    delete con_lock;
    delete con_closeDoor;
}
// signal exiters and enterers to action

void
Elevator::OpenDoors() {
    //let rider inside go out
    exit[currentfloor].Signal();
    con_lock->Acquire();
    // calculate close door num;
    int waiters = b->getFloors()[currentfloor].e[direction].Waiters();
    closeDoorNum = waiters > (capacity - occupancy)?(capacity - occupancy):waiters;
    con_lock->Release();
    //set src or srcdown , deal with problem that capacity limited
    b->getLock()->Acquire(); // Will there be any possible deadlock?
```

```
    if(direction == 1){
        b->getSrcUp()[currentfloor] = false;
    }else{
        b->getSrcDown()[currentfloor] = false;
    }
    b->getLock()->Release();
    b->getFloors()[currentfloor].e[direction].Signal();
}

void
Elevator::CloseDoors() {
    con_lock->Acquire();
    while(closeDoorNum != 0) {
        con_closeDoor->Wait(con_lock);
    }
    request[currentfloor] = false;
    con_lock->Release();
}

void
Elevator::VisitFloor(int floor) {
    // reach designated floor
    alarms->Pause(abs(floor - currentfloor) * _COSTPERFLOOR);
    currentfloor = floor;
}

bool
Elevator::Enter() {
    // judge if there has enough occupancy
    con_lock->Acquire();
    if(occupancy == capacity){  //to avoid the rider request again
        con_lock->Release();
        // to wait next time
        b->getFloors()[currentfloor].e[direction].Complete();
        return false;
    } else {
        occupancy++;
        con_lock->Release();
        b->getFloors()[currentfloor].e[direction].Complete();
        return true;
    }
}

void
Elevator::Exit() {
    con_lock->Acquire();
    occupancy--;
    con_lock->Release();
```

```
    exit[currentfloor].Complete(); //go out
}

void
Elevator::RequestFloor(int floor) {
    request[floor] = true;
    con_lock->Acquire();
    closeDoorNum--;
    if(closeDoorNum == 0) {
        con_closeDoor->Signal(con_lock);
    }
    con_lock->Release();
    exit[floor].Wait();
}

/*----------------building define-------------------------*/
Building::Building(char *debugname, int numFloors, int numElevators) {
    elevator = new Elevator(debugname, numFloors, 1);
    elevator->setBuilding(this); // Bind elevator
    name = debugname;
    srcUp = new bool[numFloors+2];
    srcDown = new bool[numFloors+2];
    floors = new Floor[numFloors+2];
    mutex = new Lock("lock for building");
    floorNum = numFloors;
}

Building::~Building() {
    delete elevator;
    delete[] floors;
    delete[] srcUp;
    delete[] srcDown;
    delete mutex;
}

void
Building::CallUp(int fromFloor) {
    mutex->Acquire();
    srcUp[fromFloor] = true;
    mutex->Release();
}


void
Building::CallDown(int fromFloor) {
    mutex->Acquire();
    srcDown[fromFloor] = true;
    mutex->Release();
```

```
}

Elevator *
Building::AwaitUp(int fromFloor)  {
     // wait for elevator arrival & going up
     floors[fromFloor].e[1].Wait();
     return elevator;
}

Elevator *
Building::AwaitDown(int fromFloor) {
     floors[fromFloor].e[0].Wait();
     return elevator;
}
```

> ***Code. 7*** Building 类与 Elevator 类的主要实现

乘客行为沿用实验指导中给出的方法如下：

```
void rider(int id, int srcFloor, int dstFloor) {
  Elevator *e;
  if (srcFloor == dstFloor)
     return;
  do {
     if (srcFloor < dstFloor) {
        building->CallUp(srcFloor);
        e = building->AwaitUp(srcFloor);
     } else {
        building->CallDown(srcFloor);
        e = building->AwaitDown(srcFloor);
     }
  } while (!e->Enter()); // elevator might be full!
  e->RequestFloor(dstFloor); // doesn't return until arrival
  e->Exit();
}
```

> ***Code. 8*** 乘客行为

实验中使用 `Alarm` 类模拟乘客于不同时间到达。

D. 电梯调度

本实验中一个关键的问题便是实现电梯的调度。

为此，我们采用这样的步骤来调度电梯：

1. 对于上行中的电梯，电梯首先将**目的层**设置为*当前楼层到最高层间有上行乘客等待的层数*与*当前梯内乘客目的地的最高层数*中的**较大者**。对于下行中的电梯，则将目的层设置为*当前楼层到最低层间有下行乘客等待的层数*与*当前梯内乘客目的地的最高层数*中的**较小者**。之后，电梯在运行到目的层的过程中若**遇到梯内乘客的目的地**或有**同向乘客等待**的层，则**停梯载客 / 卸客**，并视情况

更新目的层。

2. 当步骤 1 中的电梯找不到新的目的层时，此时电梯中无乘客。此时上行电梯检视目前层到最高层间有等待**下行**的乘客的楼层，若有，则将目的层设置为其中的最高层；下行电梯则检视目前层到最高层间有等待**上行**的乘客的楼层，若有，则将目的层设置为其中的最低层。若找到目的层，则电梯直接访问该层，途中**不停梯**。
3. 电梯**调转运行方向**，返回 步骤 1.

为调度电梯，我们在大楼类中设置了 `RunElev(int)` 方法，其具体实现如下：

```
void
Building::RunElev(int eid) {
    Elevator *elev = elevator;
    int next; // Destination
    while (true) {
        next = 0;
        mutex->Acquire();
        if (!elev->getDirection()) { // False when elevator is going down
            // Find the farest floor having riders waiting to enter or exit the
elevator in current direction
            for (int i = elev->getCurrentFloor(); i >= 1; --i) {
                if (srcDown[i] || elev->request[i]) { next = i; }
            }
        } else {
            for (int i = elev->getCurrentFloor(); i <= floorNum; ++i) {
                if (srcUp[i] || elev->request[i]) { next = i; }
            }
        }
        mutex->Release();
        if (!next && !elev->getOccupancy()) {    // No one onboard and no more
waiting rider in current direction
            if (!elev->getDirection()) { // False when elevator is going down
                // Find the farest floor having riders waiting to enter in another
direction
                for (int i = elev->getCurrentFloor(); i >= 1; --i) {
                    if (srcUp[i]) { next = i; }
                }
            } else {
                for (int i = elev->getCurrentFloor(); i <= floorNum; ++i) {
                    if (srcDown[i]) { next = i; }
                }
            }
            if (next) {
                elev->VisitFloor(next); // directly visit the floor
            }
            // Change direction
            elev->changeDirection();
            continue;
        }
        if (!elev->getDirection()) {
```

```
            for (int i = elev->getCurrentFloor(); i >= next; --i) {
                elev->VisitFloor(i);
                // If any rider wanna enter or exit on floor i, open then close
door
                if (srcDown[i] || elev->request[i]) {
                    elev->OpenDoors();
                    elev->CloseDoors();
                }
            }
        } else {
            for (int i = elev->getCurrentFloor(); i <= next; ++i) {
                elev->VisitFloor(i);
                if (srcUp[i] || elev->request[i]) {
                    elev->OpenDoors();
                    elev->CloseDoors();
                }
            }
        }
    }
}
```

> *Code. 9* 电梯调度算法

F. *threadtest.cc* 与 *main.cc* 相关修改

***main.cc* 中作出的修改**

类似于上一个实验，在 *main.cc* 里我们需要处理命令行调用 `./nachos` 时传入的参数。为此，我们定义了以下参数：

参数标记	对应变量名	参数含义
-q	int testnum	测例编号，用于进入不同的测试分支
-t	int threadNum	需要创建的并行线程数量
-n	int floorNum	电梯问题中大楼的楼层数

- 运行时附加参数 `-d t` 可以打印出相关的调试信息。
- 实验中电梯默认的容量是固定的，若要修改需要修改实验源码并重新编译。

***threadtest.cc* 中的修改**

在本次实验中，我们提供了对事件栅栏原语、闹钟原语与电梯问题的测试。对应的测试编号见下表：

测例类型	描述	测例编号
事件栅栏原语	演示多个并发线程在事件栅栏对象的作用下同步执行的情况	`-q 1`
闹钟原语	演示多个并发线程调用闹钟时的执行情况	`-q 2`
电梯问题	演示多个并发乘客线程在一座大楼、一部有限容量电梯的情况下的执行情况	`-q 3`

- 以上测例的代码实现参见 *threadtest.cc* 与其他相关数据结构对应的源文件。

三、实验结果

因电梯问题依赖于事件栅栏原语与闹钟原语，下面仅演示电梯问题的实验结果。

为测试电梯问题，我们使用以下代码随机生成指定数目的乘客线程，其中`getRandNum(int)`生成一个 1 到传入参数间的随机整数：

```
void
riderAction(int which) {
    int from = getRandNum(floorNums), to = getRandNum(floorNums);
    alarms->Pause(getRandNum(20) * 100);
    printf("[PERS] No.%d %d->%d request\n",which,from,to);
    rider(which,from,to);
    printf("[PERS] No.%d %d->%d reach\n",which,from,to);
}

void
mainThreadAction()
{
    // run elevator
    building = new Building("office buildings", floorNums, 1);
    Thread *elev = new Thread("thread for elevator");
    elev->Fork(elevatorAction, 1);
    // create riders
    Thread *t;
    for(int i = 0; i < threadNum; i++) {
        t = new Thread(getName(i + 1));
        t->Fork(riderAction, i + 1);
    }
    printf("all rider has reached dest and exit\n");
}
```

> ***Code. 10*** 乘客生成算法

容量有限情况

```
[PERS] No.1 1->2 request
[BLDG] call up in 1 floor
[BLDG] await up in 1 floor
[PERS] No.2 1->5 request
[BLDG] call up in 1 floor
[BLDG] await up in 1 floor
[PERS] No.3 1->5 request
[BLDG] call up in 1 floor
[PERS] No.4 1->6 request
[BLDG] call up in 1 floor
[BLDG] await up in 1 floor
[PERS] No.5 1->2 request
[BLDG] call up in 1 floor
[BLDG] await up in 1 floor
[ELEV] visit floor 1
[BLDG] await up in 1 floor
[ELEV] Desitination set to 1.
[ELEV] visit floor 1
[ELEV] Elevator 1 stopped at floor 1
[ELEV] on floor 1 open door
[PERS] some one enter on floor 1
[PERS] some one enter on floor 1
[PERS] some one enter on floor 1
[BLDG] call up in 1 floor
[BLDG] await up in 1 floor
[ELEV] on floor 1 close door
[PERS] request floor 2
[PERS] request floor 6
[PERS] request floor 5
[ELEV] Desitination set to 6.
[ELEV] visit floor 1
[ELEV] Elevator 1 stopped at floor 1
[ELEV] on floor 1 open door
[BLDG] call up in 1 floor
[BLDG] await up in 1 floor
[BLDG] call up in 1 floor
[BLDG] await up in 1 floor
[ELEV] on floor 1 close door
[ELEV] visit floor 1
[ELEV] Elevator 1 stopped at floor 2
[ELEV] on floor 2 open door
[PERS] exit on floor 2
[PERS] No.1 1->2 reach
[ELEV] on floor 2 close door
[ELEV] visit floor 2
[ELEV] visit floor 3
[ELEV] visit floor 4
[ELEV] Elevator 1 stopped at floor 5
[ELEV] on floor 5 open door
```

```
[PERS] exit on floor 5
[PERS] No.2 1->5 reach
[ELEV] on floor 5 close door
[ELEV] visit floor 5
[ELEV] Elevator 1 stopped at floor 6
[ELEV] on floor 6 open door
[PERS] exit on floor 6
[PERS] No.4 1->6 reach
[ELEV] on floor 6 close door
[ELEV] visit floor 6
[ELEV] Desitination set to 1.
[ELEV] visit floor 1
[ELEV] Elevator 1 stopped at floor 1
[ELEV] on floor 1 open door
[PERS] some one enter on floor 1
[PERS] request floor 5
[PERS] some one enter on floor 1
[PERS] request floor 2
[ELEV] on floor 1 close door
[ELEV] Desitination set to 5.
[ELEV] visit floor 1
[ELEV] visit floor 1
[ELEV] Elevator 1 stopped at floor 2
[ELEV] on floor 2 open door
[PERS] exit on floor 2
[PERS] No.5 1->2 reach
[ELEV] on floor 2 close door
[ELEV] visit floor 2
[ELEV] visit floor 3
[ELEV] visit floor 4
[ELEV] Elevator 1 stopped at floor 5
[ELEV] on floor 5 open door
[PERS] exit on floor 5
[PERS] No.3 1->5 reach
[ELEV] on floor 5 close door

Cleaning up...
```

> ***Fig. 1*** 3 容量电梯运载 5 人时的运行情况

可以看到有限容量（3人）的电梯在 1 楼同时有五人等待上行的情况下，只接受了 3 人入梯，完成运载后再返回 1 楼接起了其余两人并完成了运载。

随机生成乘客的情况可以采用形如 `./nachos -q 3 -t [乘客数] -n [楼高]` 的指令进行测试，因其输出过长，在此不再赘述。

四、实验中遇到的问题

> 如何实现NachOS 中线程的随机切换?

在 `system.cc` 中设置 `randomYield = TRUE;` 后 NachOS 会默认启用随机切换。

同理，在执行时加上 `-rs` 参数也能达到同样的效果。

> 如何控制电梯关门?

电梯的关门有两个条件，即：当电梯中的人都下去之后，在载客时电梯容量已满或是在该层等待的乘客都已进入时。
我们在电梯在开门之前统计一下该楼层可以进来乘客的数目，也就是取得阻塞在事件栅栏处的乘客数目和电梯容量剩余的人的数目之中的最小值，然后设置条件变量阻塞电梯线程，让最后一个进来的乘客去唤醒阻塞的电梯线程。

```
void
Elevator::OpenDoors() {
    //let rider inside go out
    exit[currentfloor].Signal(); // 电梯中的人出去
    con_lock->Acquire();
    //统计数目
    int waiters = b->getFloors()[currentfloor].e[direction].Waiters();
    closeDoorNum = waiters > (capacity - occupancy)?(capacity - occupancy):waiters;
    con_lock->Release();
    .....
}

void
Elevator::CloseDoors() {
    con_lock->Acquire();
    while(closeDoorNum != 0){// 使用条件变量阻塞
        con_closeDoor->Wait(con_lock);
    }
    request[currentfloor] = false;
    con_lock->Release();
}

void
Elevator::RequestFloor(int floor) {
    request[floor] = true;
    con_lock->Acquire();
    closeDoorNum--;
    if(closeDoorNum == 0){ //最后一个进来的乘客 唤醒等待关门的电梯
        con_closeDoor->Signal(con_lock);
    }
    con_lock->Release();
    exit[floor].Wait();
}
```

> 什么时候设置指示当前楼层上 / 下行方向有乘客等待的信号灯（实验中的 `srcUp` 与 `srcDown` 布尔数组）的亮灭?

在电梯容量无限的情况下,我们将电梯对信号灯数组的灭设置放在 `closeDoor` 方法中，如下:

```
void
Elevator::CloseDoors() {
     con_lock->Acquire();
     while(closeDoorNum != 0){
          con_closeDoor->Wait(con_lock);
     }
   b->getLock()->Acquire(); // 设置该楼层提出进电梯请求的信号
     if(direction == 1){
          b->getSrcUp()[currentfloor] = false;
     } else {
          b->getSrcDown()[currentfloor] = false;
     }
     b->getLock()->Release();
     request[currentfloor] = false;
     con_lock->Release();
}
```

但是这样在容量有限的情况下会有问题：有可能乘客在因电梯已满没能进入电梯后，调用 `Building` 的 `CallUp/CallDown` 方法再次设置信号灯，然后阻塞在 `awaitUp/awaitDown` 方法中,之后电梯才执行了 `closeDoor` 方法，覆盖了没有进来的乘客的请求，于是造成 `srcUp` 与 `srcDown` 布尔数组丢失信号，电梯不会再次返回该层。

于是我们将这个设置放到了开门的时刻以解决此问题，代码如下：

```
void
Elevator::OpenDoors()
{
     ...
     //set src or srcdown , deal with problem that capacity limited
     b->getLock()->Acquire(); // Will there be any possible deadlock?
     if(direction == 1){
          b->getSrcUp()[currentfloor] = false;
     }else{
          b->getSrcDown()[currentfloor] = false;
     }
     b->getLock()->Release();
     //接该楼层的人
     b->getFloors()[currentfloor].e[direction].Signal();
}
```

> 如何避免闹钟中尚有线程处在等待状态，但 NachOS 会因系统中没有活跃进程而停机的问题?

可以通过修改 NachOS 的机器定义或者创建一个守护线程实现。

守护线程可由闹钟原语在发现系统中已无其他活跃线程但闹钟中还有尚未到达唤醒时间的线程时创建。

五、实验收获与总结

本次实验为整个学期的操作系统实验课程划上了句号。通过本次实验，我们利用第二次实验中完成的锁机制与条件变量机制实现了更为强力的事件栅栏同步原语，并利用 NachOS 中模拟的 Timer 实现了闹钟原语，从而使得线程的调度控制成为可能。最后，我们利用了事件栅栏原语与闹钟原语成功模拟了一座大楼、一部有限容量电梯、若干乘客的情况下的电梯问题，顺利完成了整个实验。

在本次实验的过程中，我们深刻地体会到了有效的同步机制的设计与实现对于编写正确的并发程序的重要性。同步机制不仅提供了对多线程并行程序的共享资源的访问保护，更重要的是类似于事件栅栏这一强有力的同步原语大大减少了并发编程的困难度，也使得编程过程中逻辑错误出现的可能性大为降低。而闹钟原语则使得对线程的基本调度成为可能。

总的说来，这次实验可以看成前两次实验乃至操作系统课程整个学期大部分重难点知识的集大成者，通过这次实验我们对于操作系统中并发编程的同步问题、互斥问题、死锁问题、多任务的调度及中断机制都有了更为深刻的了解。同时，这次实验最大的意义之一在于实现了书本知识与编程实践之间的转化。虽然我们为本次实验提供的解法必然存在着诸多可以改良的地方，但随着实践经验的进一步积累，相信遇到类似的问题变体的时候我们会有更好的想法。