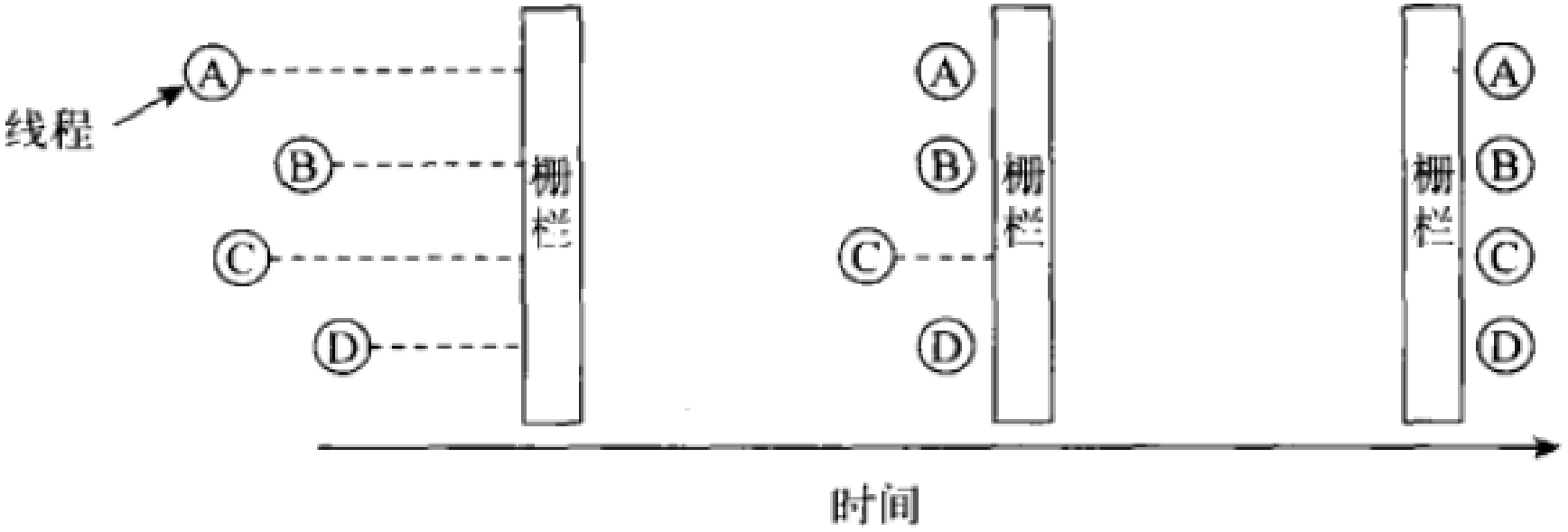
实验3：Nachos下的线程编程

第三次实验简介

- 任务：使用Nachos线程系统解决电梯同步问题
  - 在此之前需要先实现两个原语“栅栏”和“闹钟”，这两个原语将有助于电梯同步问题解决方案的实现
- 目标：提高两方面的技能
  - 编写正确的并发程序
  - 应对并发编程中常见的陷阱：竞争、死锁、饥饿等
- 注意：
  - 第三次实验将使用第二次实验中实现的同步原语，请务必确保使用到的同步原语是正确的

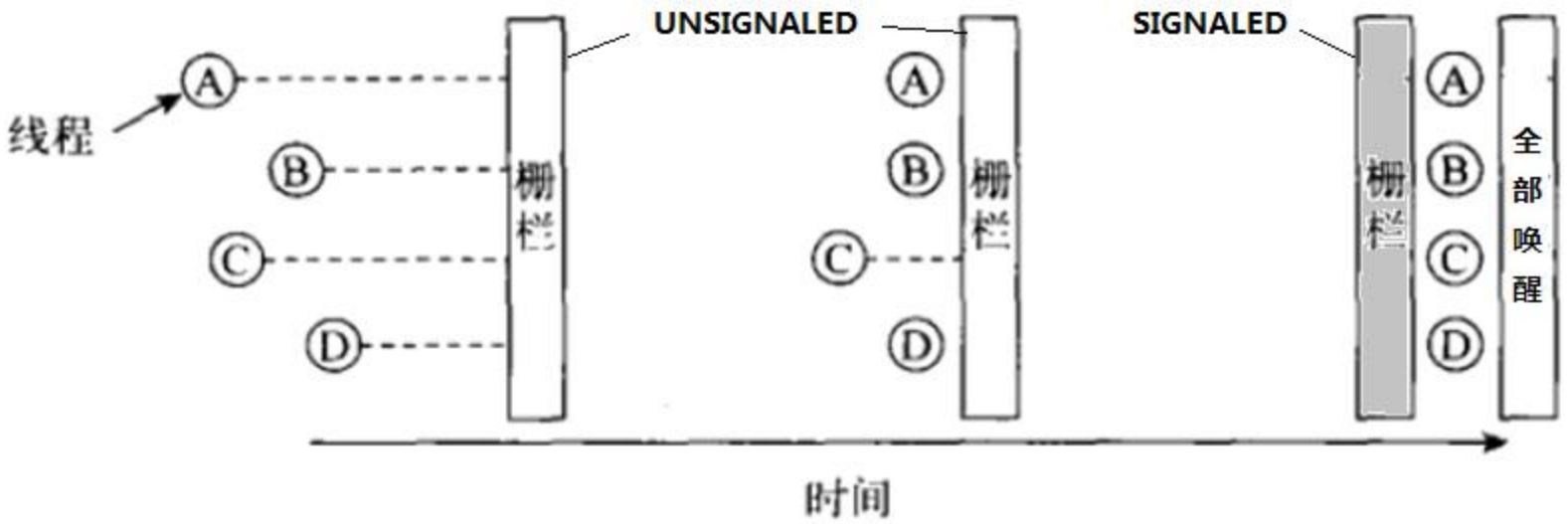
什么是栅栏

- 有时候一组进程协同完成一个问题，我们需要所有进程都到同一个地点汇合后再一起向前推进
- “栅栏”就是一个路障，先到达栅栏的进程必须停下来等待，知道所有进程都到达，栅栏除去，大家一起向前推进



实现栅栏的关键是同步

- 一个运转周期里需要几次同步？
  - 线程等待同步
    - 线程先后到达，先到达的等待直到栅栏被放开(SIGNALED)；
  - 线程唤醒同步
    - 某线程放开栅栏，向等待的线程发出广播，自己陷入等待直到全部被唤醒，然后重置栅栏；
    - 等待的线程先后被唤醒，先唤醒的需要等待直到全部被唤醒



实现栅栏类EventBarrier

- nachos-labs.pdf 章节3.3.1提供了接口
  - void Wait() – 用于线程等待同步：由需要同步的线程调用，调用者陷入等待直到被唤醒。若栅栏已处于放开状态，则直接返回
  - void Signal() – 由控制栅栏的线程调用，唤醒当前正在等待的所有线程，自己陷入等待直到线程已被全部唤醒，苏醒后重置栅栏状态。
  - void Complete() – 用于线程唤醒同步：由需要同步的线程调用，从Wait返回（被唤醒）后应接着调用Complete，随后陷入另一种等待直到所有线程都被唤醒。
  - int Waiters() – 返回当前处于等待（或尚未被唤醒）的线程数量，用来标记线程是否已被全部唤醒。

实现EventBarrier的注意事项

- 请自己创建EventBarrier.h和EventBarrier.cc文件
- 可以使用锁，条件变量（推荐），信号量的任意组合，但不允许直接采用“禁止中断”（请思考为什么）；
- 调用Wait时，根据当前栅栏的状态，执行应有所不同；
- 调用Complete时，根据当前等待线程的数量，执行应有所不同；
- 若有线程从Complete返回后又立即调用Wait，则该线程应陷入等待直到栅栏再次放开（即等待栅栏再次变为Signaled状态）；
- 实现后应写一个简单的测试程序测试效果。

实现闹钟原语的预备知识

- **Nachos的计时机制**
  - ◆在虚拟机上通过中断模拟实现，一个Tick（滴答）并不是由硬件产生的，而是模拟的；
  - ◆虚拟机用一条指令的执行时间表示一个滴答，因此模拟的时间无法与现实世界的时间准确对应；
  - ◆请仔细阅读machine/timer.h, machine/timer.cc, machine/interrupt.h, machine/interrupt.cc, threads/system.h, threads/system.cc, threads/main.cc

实现闹钟类Alarm的基本思路

- 实现闹钟设定Alarm::Pause(int howLong)，
  - 调用该函数的线程将睡眠howLong个时间单位（具体单位时间可自拟）；
  - 函数执行时线程应把自己和闹醒时刻存入队列中然后陷入睡眠，此操作过程应为原子操作（可以使用“禁止中断”实现）；
- 实现闹钟唤醒（函数名自拟）
  - 每当时间中断发生时调用（请自己找到时间中断处理函数）
  - 函数执行时，遍历已存入队列的闹钟记录，判断是否已到时，如果是则移除该记录并唤醒相关线程（使之转为runnable状态），此操作过程应为原子操作（可以使用“禁止中断”实现）；
- 请自己创建Alarm.h和Alarm.cc文件
  - 与系统时间有关的操作实例可参考interrupt.cc的代码

实现Alarm的特别注意事项

1. 如果系统中没有线程处于runnable状态，哪怕有线程在等待闹钟，nachos也将自动退出。为了避免这种情况的发生，你可以：
   - 改写 machine目录里的代码（合情合理，但是不推荐）；
   - 或者，在Alarm::Pause(int howLong)函数中，当发现自己将成为唯一一个等待闹钟的线程时，创建一个“打酱油”线程，该线程反复检查当前有几个等待闹钟的线程，若为零则结束，若不为零则切换（比较猥琐，但是推荐）；
2. 初始的nachos代码只有在使用“-rs”选项的情况下才会启动时间中断，可对 threads/system.cc稍做修改使得不论“-rs”选项使用与否，时间中断都会被启用，但是，不允许修改timer.cc或timer.h！

解决电梯的同步问题

- 乘坐电梯过程中的同步举例（使用栅栏）
  - 给某个楼层的上（下）方向分配一个栅栏 b，先后出现在某个楼层的乘客希望上（下）行，陷入等待（乘客调用 b.Wait()），直到上行的电梯到达（假设此时电梯内无乘客打算出电梯，电梯开门并调用 b.Signal()），乘客开始进电梯（乘客调用 b.Complete()），电梯关门。
- 尽可能多地捕捉乘坐电梯场景中类似的同步问题，利用已有的同步机制（锁，信号量，条件变量，栅栏）解决它们。

模拟简单的电梯场景

- 一座楼 + 一个无限容量电梯 + 若干乘客
  - 一座楼：主线程创建一个静态Building对象
  - 一个电梯：就是一个独立的动态线程（由Building对象创建并初始化一个电梯对象，然后主线程创建一个新线程使电梯对象按照设计好的规则运行）
  - 若干乘客：每个乘客是一个独立的动态线程，由主线程创建并使之按设计好的规则运行

实现Building类

- 基本框架定义已在Elevator.h头文件提供（在课程网站lab3-header.tar.gz文件内）
- 已定义的提供乘客线程调用的接口不可更改:
  - CallUp(int fromFloor) // 在某层按上行按钮
  - CallDown(int fromFloor) // 在某层按下行按钮
  - Elevator*AwaitUp(int fromFloor) // 在某层等待上行电梯
  - Elevetor*AwaitDown(int fromFloor) // 在某层等待下行电梯
- 自行定义所需的类成员变量或函数
  - 思考Building类里需要多少个，什么类型的同步变量
  - 思考需要设置哪些信息以控制电梯线程的运行

实现Elevator类

- 基本框架定义已在Elevator.h头文件提供
  - // 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 destinationFloor
- 自行定义所需的类成员变量或函数
  - 思考Building类里需要多少个，什么类型的同步变量
  - 思考需要设置哪些信息以控制电梯线程的运行
  - 设计一个成员函数，模拟电梯的行为，可用闹钟模拟电梯移动间隔

模拟乘客的行为

```
1.  void rider(int id, int srcFloor, int dstFloor) { // 可利用闹钟每隔一段时间
2.      Elevator *e;                                 // 乘客线程调用此函数
3.      if (srcFloor == dstFloor)                    // 参数可随机生成
4.          return;
5.      do {
6.          if (srcFloor < dstFloor) {
7.              building->CallUp(srcFloor);          // a1.按下上行按钮
8.              e = building->AwaitUp(srcFloor);     // b1.等待上行电梯
9.          } else {
10.             building->CallDown(srcFloor);        // a2.按下下行按钮
11.             e = building->AwaitDown(srcFloor);   // b2.等待下行电梯
12.         }
13.     } while (!e->Enter());                       // elevator might be full! 3.进入电梯
14.     e->RequestFloor(dstFloor);                   // doesn‘t return until arrival 4.选定楼层
15.     e->Exit();                                   // 5.离开电梯
16. }
```

考虑更复杂的场景

- 一座楼 + 一个有限容量的电梯 + 若干乘客
  - 若电梯已满，乘客需继续等待
- 一座楼 + N个有限容量的电梯 + 若干乘客
  - 选做
  - 需要设计一个机制，使得当乘客按下上行（或下行）按钮时，能选择一个合适的电梯提供服务

本次实验提交内容

- 本次实验应提交的内容和应出现的位置（~表示各组长的主目录）
  - ~/nachos/3/Makefile.common
  - ~/nachos/3/system.cc
  - ~/nachos/3/system.h
  - ~/nachos/3/main.cc
  - ~/nachos/3/threadtest.cc
  - ~/nachos/3/EventBarrier.h
  - ~/nachos/3/EventBarrier.cc
  - ~/nachos/3/Alarm.h
  - ~/nachos/3/Alarm.cc
  - ~/nachos/3/Elevator.h
  - ~/nachos/3/Elevator.cc
  - ~/nachos/3/nachos03.doc
- **本次实验提交截止时间：2018年6月16日中午12：00**