6、管程

内容

- 信号量机制的问题
- 管程的概念
- 引入条件变量的管程
- 管程的实现方法
- 实现哲学家就餐问题的例子
- 常用操作系统的同步机制

信号量机制的问题

- 优点
  - 程序效率高、编程灵活
- 问题
  - 需要程序员实现，编程困难
  - 维护困难、容易出错
    - ✓wait/signal位置错
    - ✓wait/signal不配对
- 解决方法
  - 管程（1970s, Hoare和Hansen）
  - 由**编程语言**解决同步互斥问题，而不是程序员

信号量：**分散式**
管　程：**集中式**

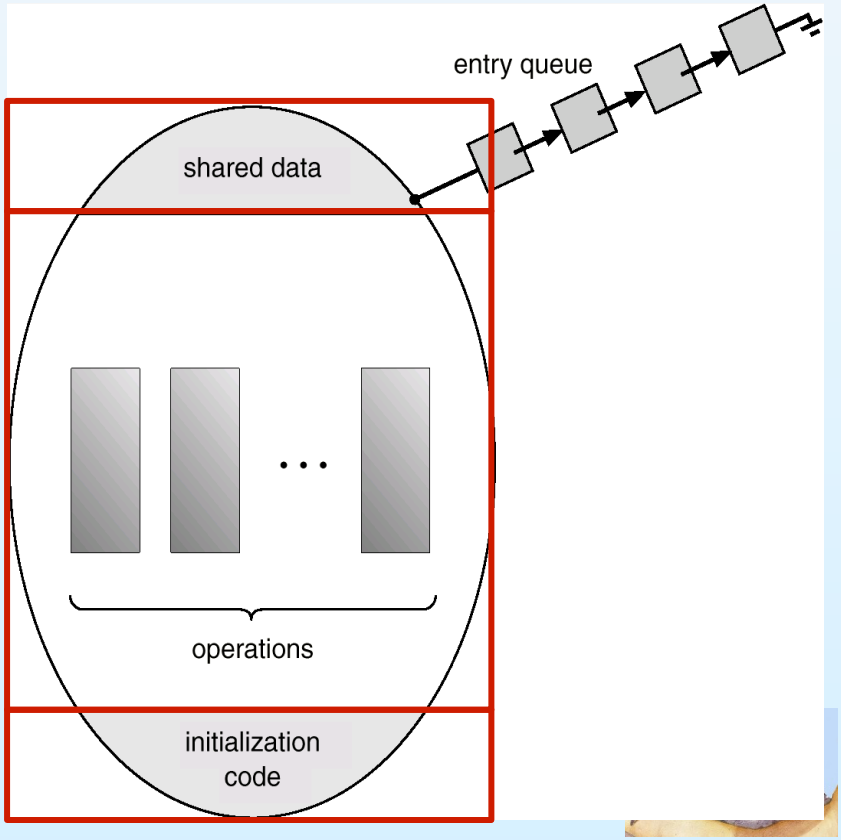
管程Monitors

Hansen的管程定义

一个管程定义了一个**数据结构**和能为并发进程所执行（在该数据结构上）的**一组操作**，这组操作能**同步进程**和**改变管程中的数据**

- 高级同步构建类型
- 管程是对提供线程安全机制的高度抽象
- 任一时刻在管程中只有一个线程能运行

```
monitor monitor-name
{
    // variable declarations
    public entry p1(…) {…}
    public entry p2(…) {…}
        …
    Initialization_code(…){…}
}
```



管程功能

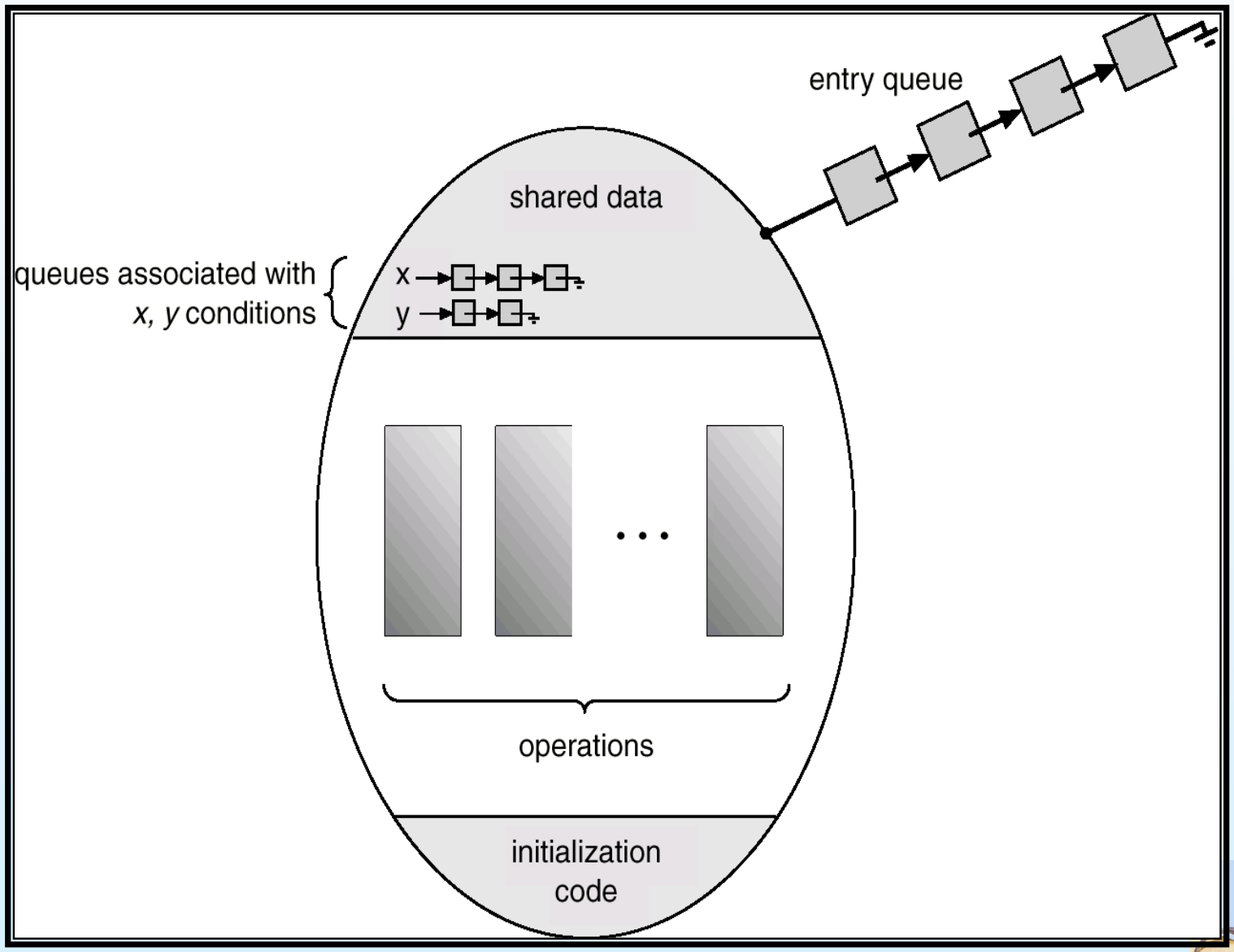
- 互斥
  - 管程中的变量只能被管程中的操作访问
  - 任何时候只有一个进程在管程中操作
  - 类似临界区
  - 由编译器完成
- 同步
  - 条件变量
  - 唤醒和阻塞操作

- 判断：在管程中加入条件变量，以及基于条件变量的唤醒和阻塞操作可以为进程提供同步机制。

管程功能

- **condition x, y**;
- 条件变量的操作
  - x.wait(): 进程阻塞直到另外一个进程调用x.signal()
  - x.signal()：唤醒另外一个进程

条件变量

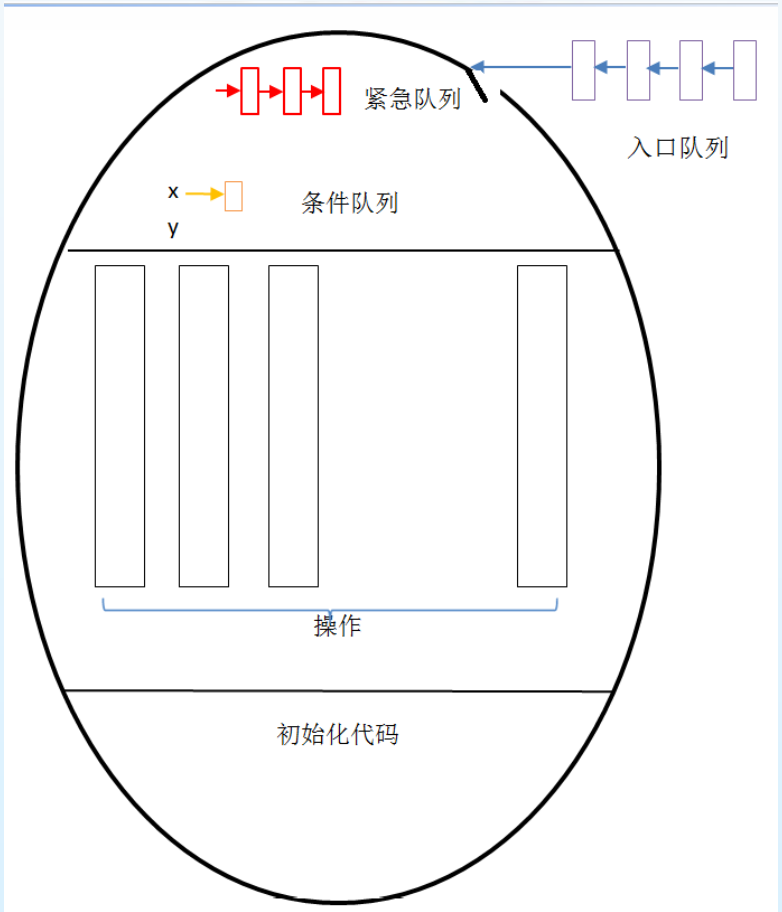

entry queue
shared data
queues associated with
x, y conditions
x
y
operations
initialization
code

条件变量问题

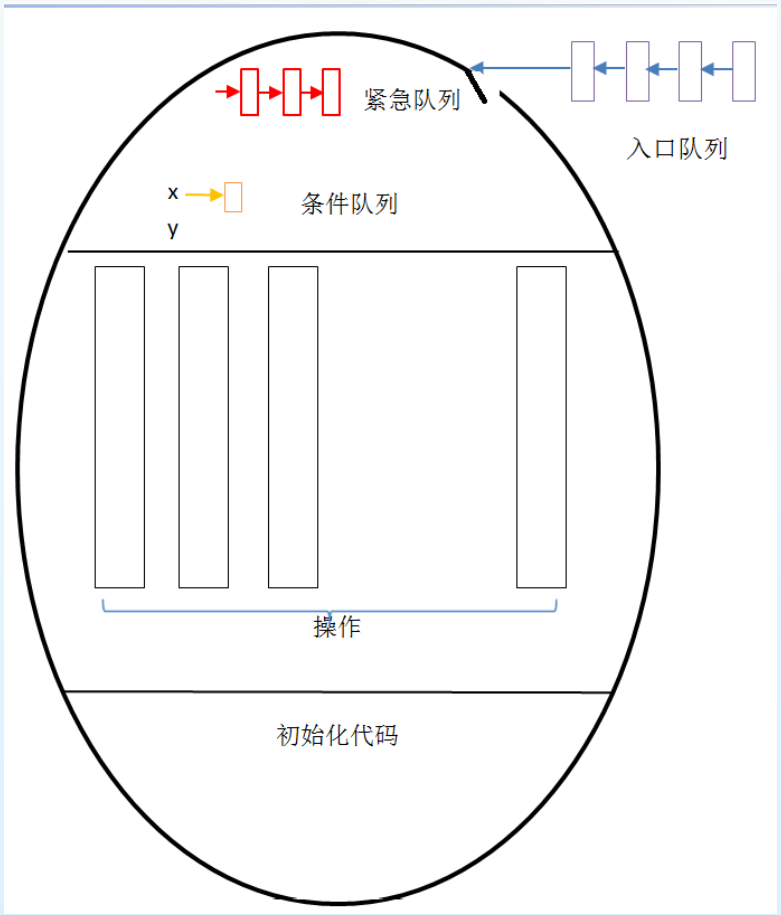
- 管程内可能存在不止1个进程
  - 如：进程P调用signal操作唤醒进程Q后，此时管程内有P和Q两个进程
- 存在的可能
  - P等待直到Q离开管程（Hoare）
  - Q等待直到P离开管程（Lampson & Redll，MESA语言）
  - P的signal操作是P在管程内的最后一个语句，然后Q开始运行 (Hansen，并行Pascal)

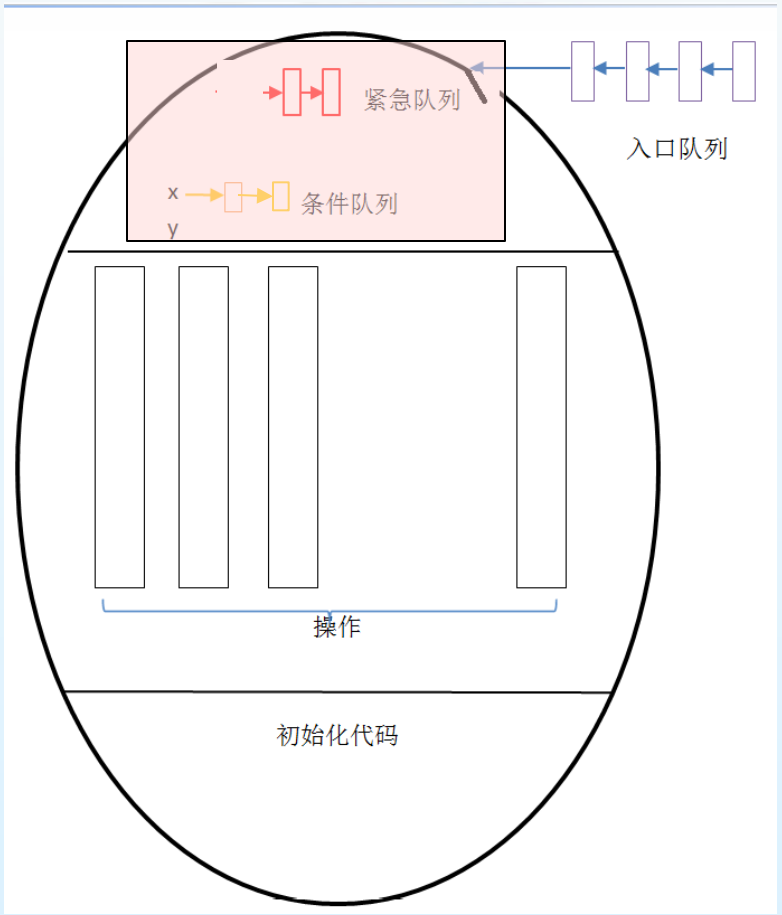
Hoare管程

- 进程互斥进入管程
  - 如果有进程在管程内运行，管程外的进程等待
  - **入口队列**：等待进入管程的进程队列
- 管程内进程P唤醒Q后
  - P等待，Q运行
  - P加入**紧急队列**
  - 紧急队列的优先级高于入口队列

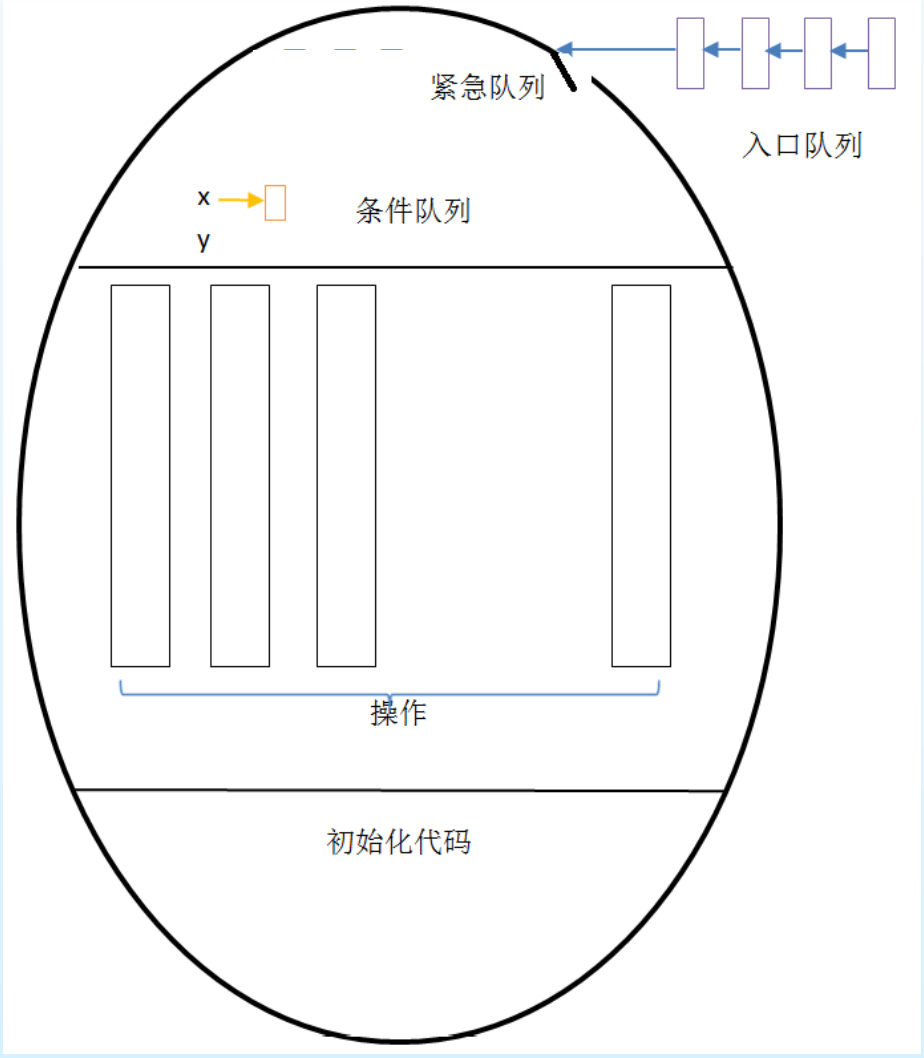


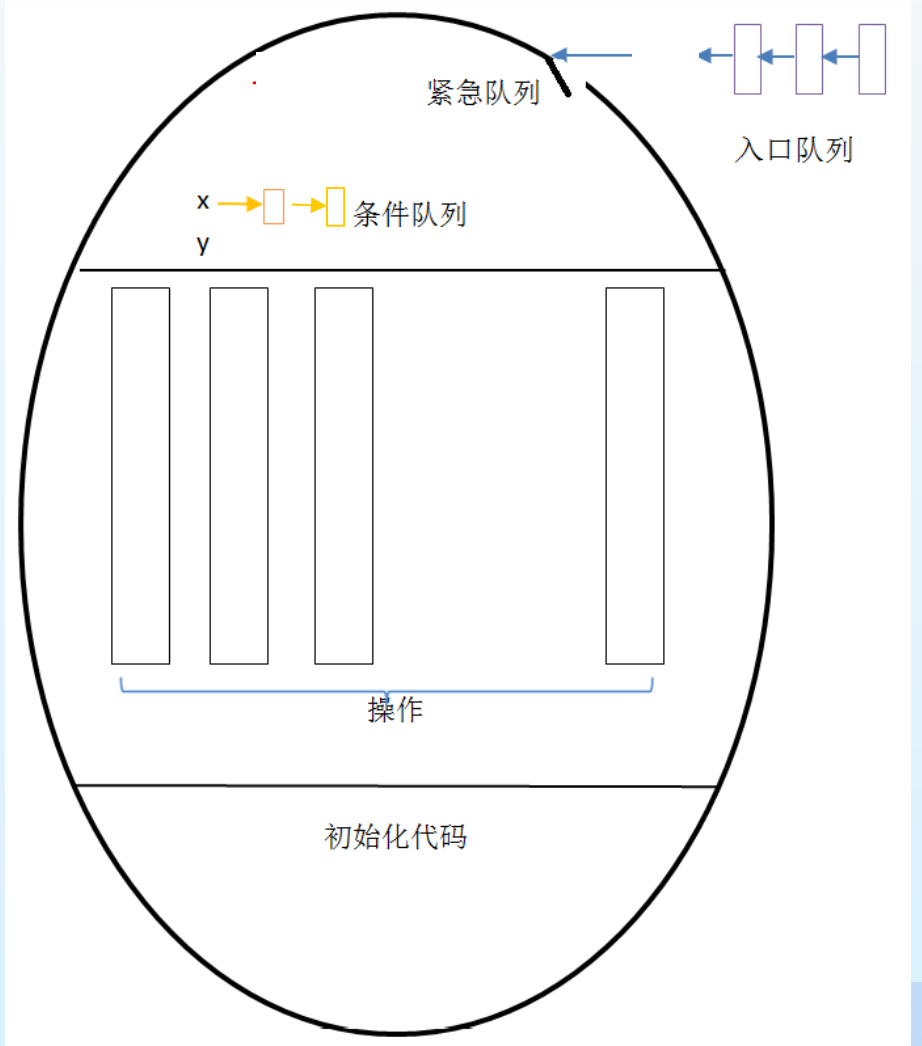
紧急队列非空





紧急队列为空





Hoare管程

- condition x;
- x.signal()
  - x的条件队列空：空操作，执行该操作进程继续运行
  - x的条件队列非空：唤醒该条件队列的第一个等待进程，执行唤醒后的进程，而该操作进程进入紧急队列

哲学家就餐Hoare管程解决方案

```
monitor DP
{
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];

    initialization_code() {
            for (int i = 0; i < 5; i++)
            state[i] = THINKING;
        }

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self[i].wait();
    }

    void putdown (int i) {
        state[i] = THINKING;
            // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test (int i) {
        if ( (state[(i + 4) % 5] != EATING)
     &&(state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
            state[i] = EATING ;
            self[i].signal() ;
        }
    }
}
```

共享变量

初始化

操作

哲学家就餐问题的管程解决方案

- 每个哲学家按照以下的顺序轮流调用操作 pickup() 和putdown()

```
dp.pickup (i)

    EAT

dp.putdown (i)
```

6、同步实例

Linux同步机制

- 使用禁止中断来实现短的临界区

- 自旋锁(spinlock)
  - 不会引起调用者阻塞
- 互斥锁(Mutex)
- 条件变量(Condition Variable)
- 信号量(Semaphore)

Windows同步机制

- 事件(Event)
  - 通过通知操作的方式来保持线程的同步
- 临界区(Critical Section)
- 互斥锁(Mutex)
- 自旋锁(Spinlock)
- 信号量(Semaphore)