进程并发控制：互斥与同步

薛瑞尼

计算机科学与工程学院

16/3/18

回顾和问题

- 进程状态模型、调度方法
  - 等待I/O就绪
  - 时间片

  } 阻塞/挂起

- 非自愿，被动暂停

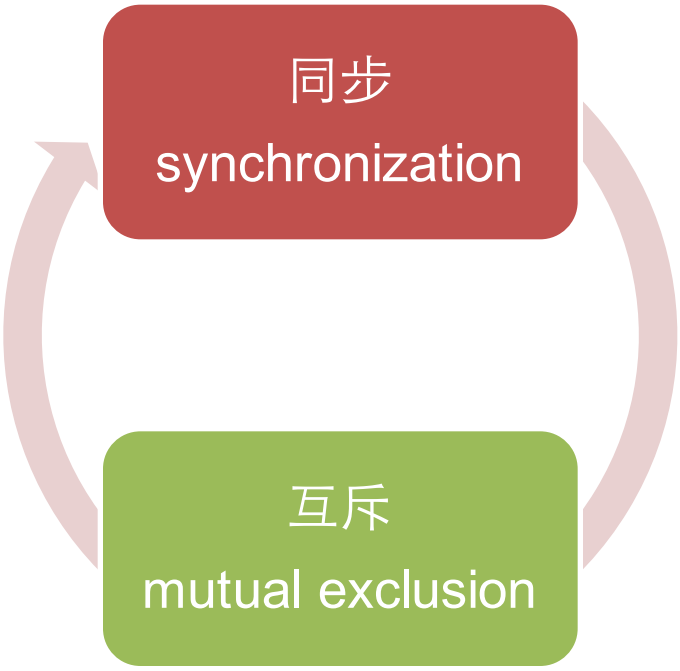
- 如无I/O等待或时间片未到，是否会主动“阻塞/挂起”？

生活中的例子

- 十字路口，理发店、银行、网球
- 为什么会出现“规则和秩序”？
  - 个体自主性：异步（车辆/行人/顾客）
  - 资源稀缺性：独占（十字路口/理发师）
  - 任务需合作：协作

进程/线程的并发控制

- 进程/线程是计算机中的独立个体：异步性（并发性）
- 资源是计算机中的稀缺个体：独占性（不可复用性）
- 进程/线程协作完成任务



- 并发控制：进程/线程在推进时的相互制约关系

并发执行进程能有效地共享资源和相互合作，并按一定顺序执行。

例子

- 某对象obj拥有私有变量count，初值为0
- 以及一个函数

```
public void inc() {
   count = count + 1;
}
```

- 若两个线程T1，T2同时执行obj.inc()

执行过程

- 假设线程的底层执行为

```
LOAD reg, count
ADD reg, 1
STORE count, reg
```

- 可能的执行过程

T1

```
0   LOAD r1, count
1   ADD r1, 1
    ..
    ..
    ..
1   STORE count, r1
```

T2

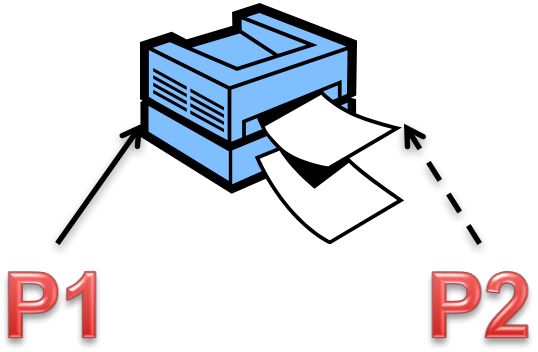
```
    ..
    ..
0   LOAD r2, count
1   ADD r2, 1
1   STORE count, r2
    ..
```

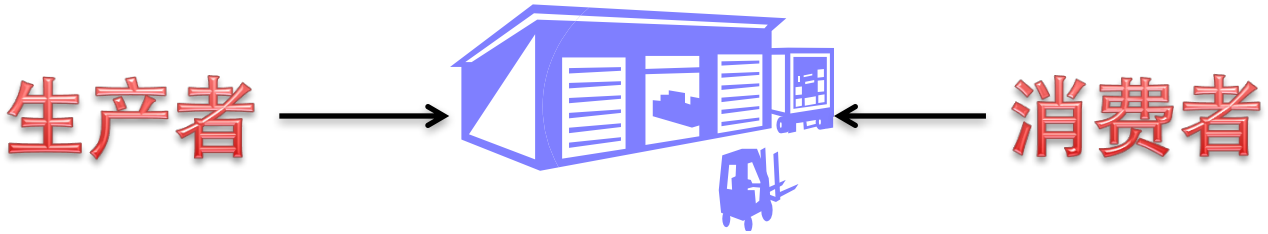
Race condition

- A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.

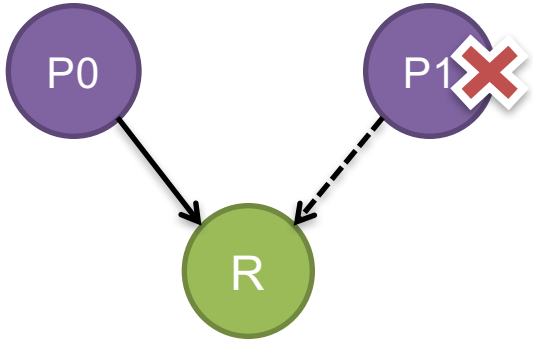
基本概念

- 进程间的制约关系
  - 间接制约：资源共享 →互斥
  - 直接制约：进程合作 →同步
- 临界资源（Critical Resource）
  - 一次仅允许一个进程访问的资源
- 临界区（Critical Section）
  - 进程中访问临界资源的代码段

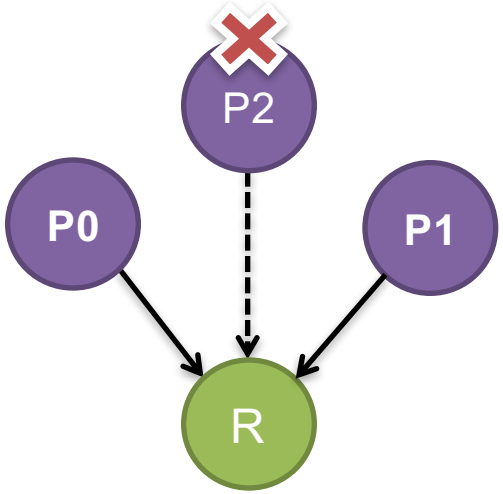




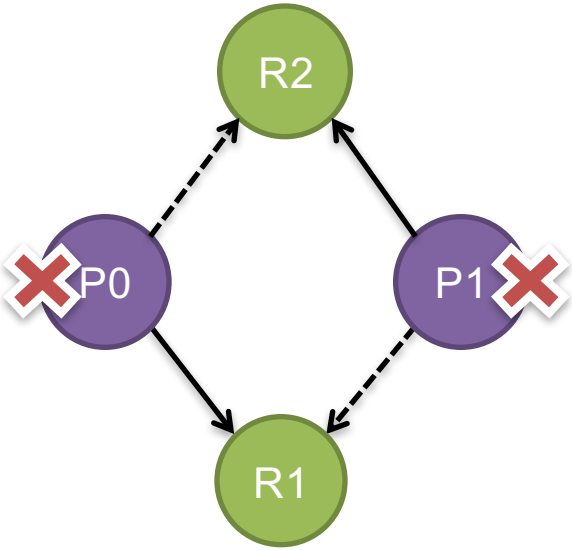
忙等、饥饿、死锁



忙等
busy waiting

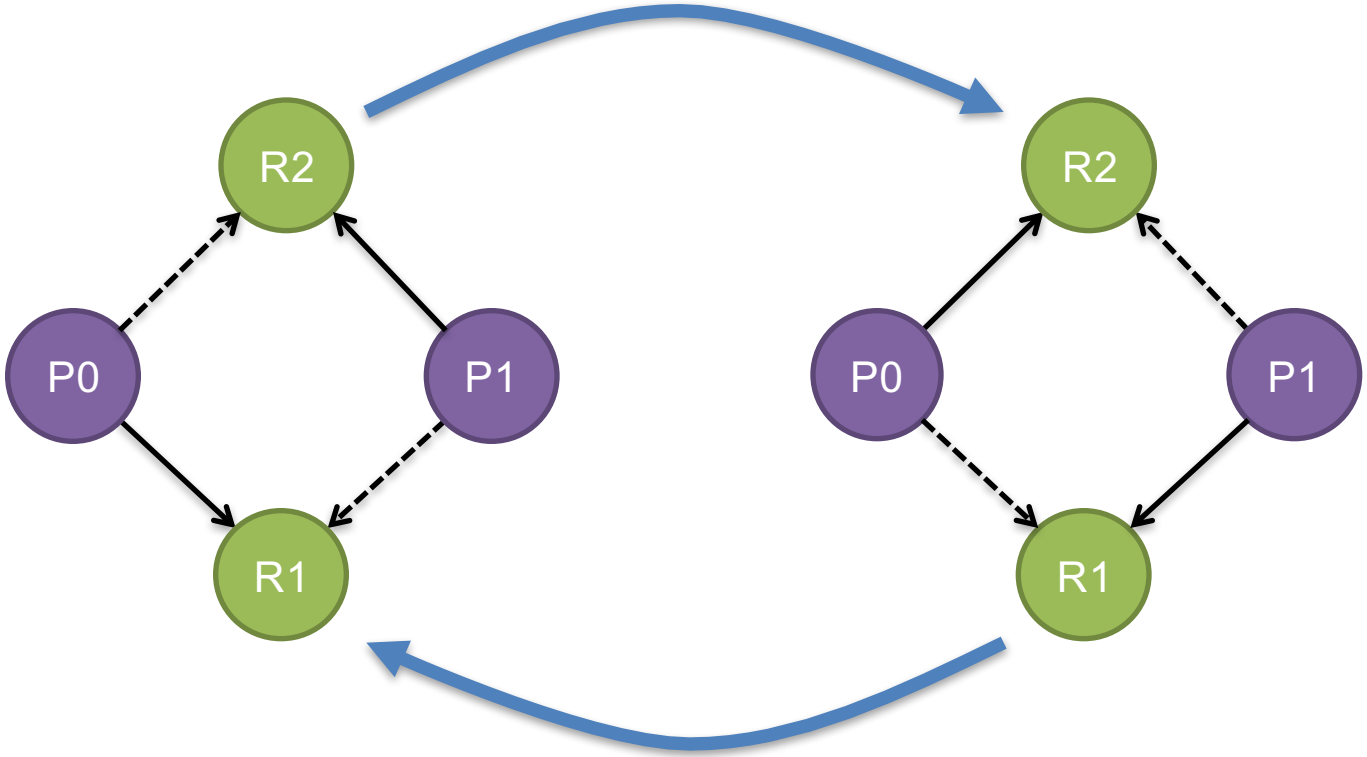


饥饿
starvation



死锁
deadlock

活锁（livelock）



A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.

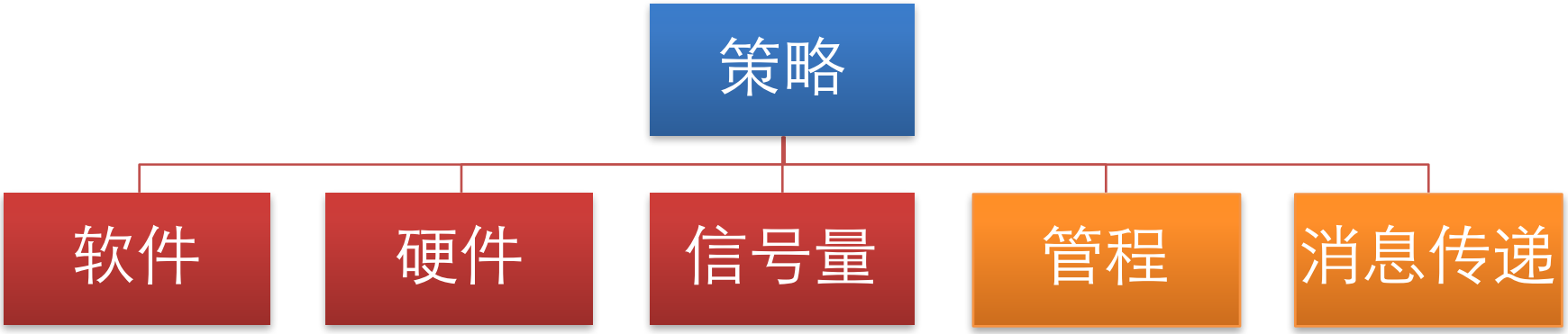
同步原则

- 空闲让进
- 忙则等待
- 有限等待
- 让权等待

效率

公平

同步的解决策略



软件方法

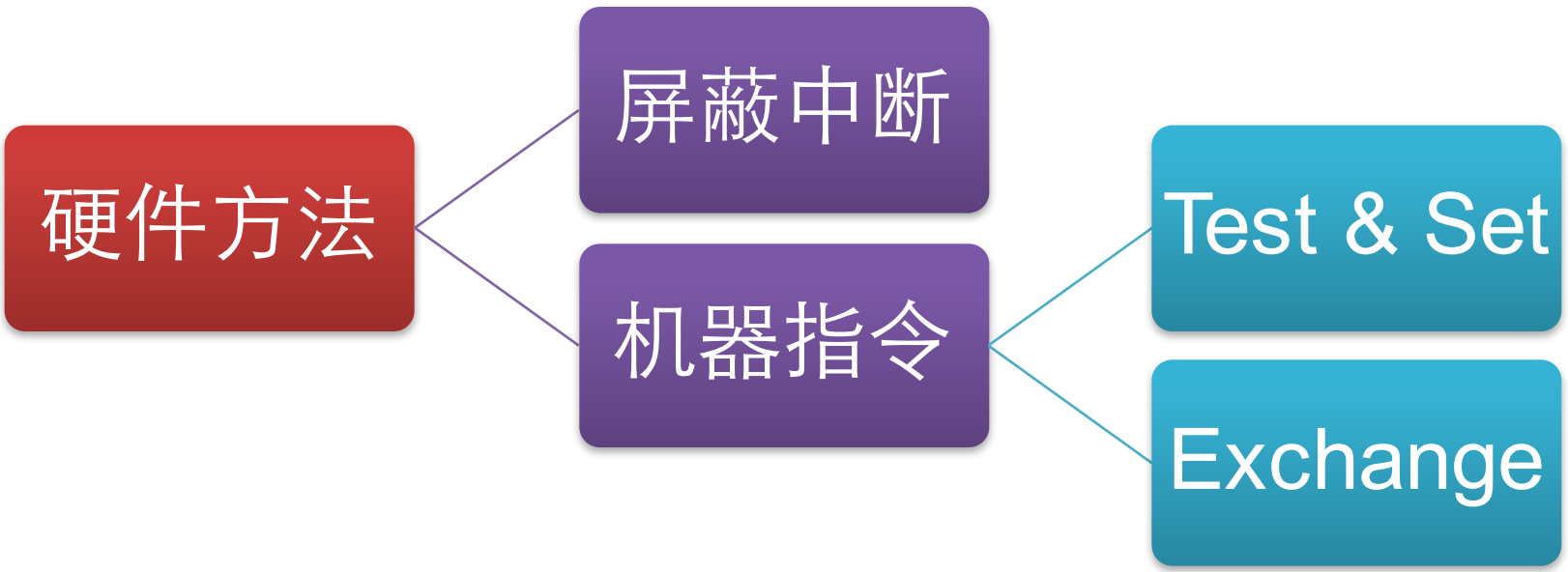
- 进程通过执行相应的程序指令，实现与其他进程的互斥与同步
  - 交替进入，崩溃失效
  - 系统的开销大（忙等），正确性难保证；
  - Dekker，Peterson算法（详见教材）

```
1: var turn: 0..1; {* 共享全局变量 *}
2:
3: P0
4:
5: while turn != 0 do {nothing};
6: <临界区>
7: turn := 1;
```

```
1: P1
2:
3: while turn != 1 do {nothing};
4: <临界区>
5: turn := 0;
```

思考：互斥进程数>2?

硬件方法



屏蔽中断

- 进程运行中屏蔽中断，则不会出现进程切换
  - 确保互斥执行
- 缺点：
  - 系统无法响应外部请求
  - 无法接受异常，处理系统故障
  - 无法切换进程→性能下降
  - 不支持多处理机

```
1: while(1) {
2:     <屏蔽中断>;
3:     <临界区>;
4:     <启用中断>;
5:     <其余部分>;
6: }
```

机器指令

- 在多处理器环境中，几个处理器共享访问公共主存；
- 处理器表现出一种对等关系，不存在主／从关系；
- 处理器之间没有支持互斥的中断机制；
- 处理器的设计者提出了一些机器指令，用于保证两个动作的原子性
  - 如在一个周期中对一个存储器单元的读和写；
  - 这些动作在一个指令周期中执行，不会被打断，不会受到其他指令的干扰。

Test & Set

- 测试某个变量的值，如果为0，则置1，并返回当前值

```
1: function testset(var i:integer): boolean
2: begin
3:   if i = 0 then
4:     begin
5:       i := 1;
6:       return true;
7:     end
8:   else return false;
9: end
```

- X86: TEST

示例

```
1:  program mutualexclusion;
2:    const n＝...; {* 进程数 *}
3:    var bolt: integer;
4:  procedure P(i:integer);
5:  begin
6:    repeat
7:      repeat { do no-op } until testset(bolt); {* 当bolt为0时，进入临界区*}
8:      <critical section>;
9:      bolt := 0;
10:     <remainder>;
11:   forever
12: end;
13: begin {* main program *}
14:   bolt := 0;
15:   parbegin
16:     P(1); P(2); ... P(n);
17:   parend
18: end
```

思考：如果testset不是原子操作?

Test & Set的其它定义

- An instruction used to write to a memory location and return its old value as a single atomic (i.e., non-interruptible) operation. Typically, the value 1 is written to the memory location.

```
int TestAndSet(int* lockPtr) {
    int oldValue;
    oldValue = *lockPtr;
    *lockPtr = 1;

    return oldValue;
}
```

```
volatile int lock = 0;

void Critical() {
    while (TestAndSet(&lock) == 1);
    critical section
    lock = 0;
}
```

```
void acquire(int* lock) {
  while (TestAndSet(lock) == 1);
}
```

```
void release(int* lock) {
  *lock == 0;
}
```

Exchange指令

- 原子性地交换寄存器和内存的值

```
1: procedure exchange(var r: register; var m: memory);
2:    var temp;
3:    begin
4:       temp := m;
5:       m := r;
6:       r := temp;
7:    end
```

- X86: XCHG

示例

```
1:  program mutualexclusion;
2:  const n=...; {*number of processes*}
3:  var bolt: integer;
4:  procedure P(i: integer);
5:  var key: integer;
6:  begin
7:    repeat
8:      key := 1;
9:      repeat exchange(key, bolt) until key=0;
10:     <critical section>;
11:     exchange(key, bolt);
12:     <remainder>;
13:   forever
14: end;
15: begin {*main program*}
16:   bolt:= 0;
17:   parbegin
18:     P(1); P(2); ... P(n);
19:   pare
20: end
```

思考：如果exchange不是原子操作?

机器指令

优点

- 支持多处理机
- 简单，易证明
- 支持多临界区

不足

- 忙等现象
- 饥饿现象
- 死锁现象

信号量(Semaphore)

- 交通信号灯
- 原理：多进程通过信号传递协调工作，根据信号指示停止执行（阻塞等待）或者向前推进（唤醒）。
- 信号：信号量s
  - +：资源数量
  - -：排队数量

- 原语：
  - wait(s)：等待信号，并占有资源
  - signal(s)：释放资源，并激发信号

整数型信号量

- wait(s)
  - while s <= 0:
    - do no-op
  - s := s – 1
- P操作
  - *p*robeer te verlagen:

    try to reduce
- signal(s)
  - s := s + 1
- V操作
  - *v*erhogen: increase

Edsger Wybe Dijkstra
(May 11, 1930 – August 6, 2002)
Turing Award@1972

记录型信号量

- 一种不存在“忙等”现象的进程同步机制。

定义

```
1: type semaphore=record
2:   value: integer;
3:   L: list of process;
4: end
```

wait能否将减法后置?

wait

```
1: procedure wait(S)
2:   var S: semaphore;
3:   begin
4:     S.value := S.value - 1;
5:     if S.value < 0 then
6:       begin
7:         block(process);
8:         enqueue(process, S.L);
9:       end
10:  end
```

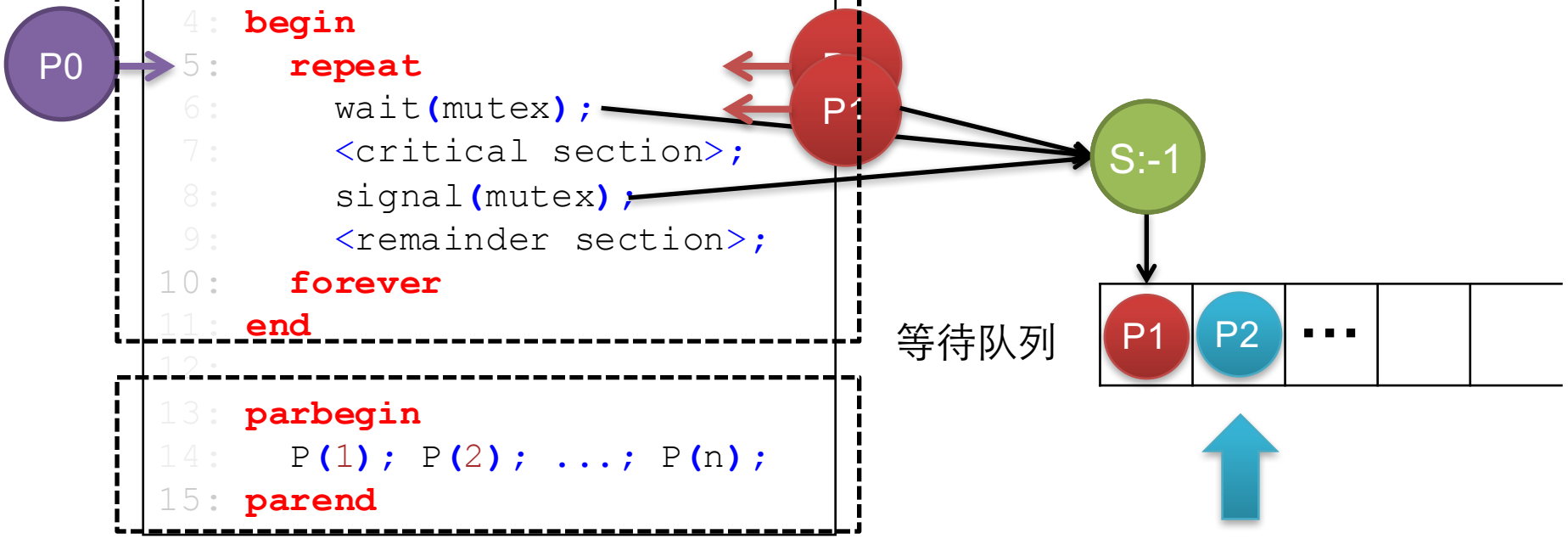
这个定义正确的前提是什么?

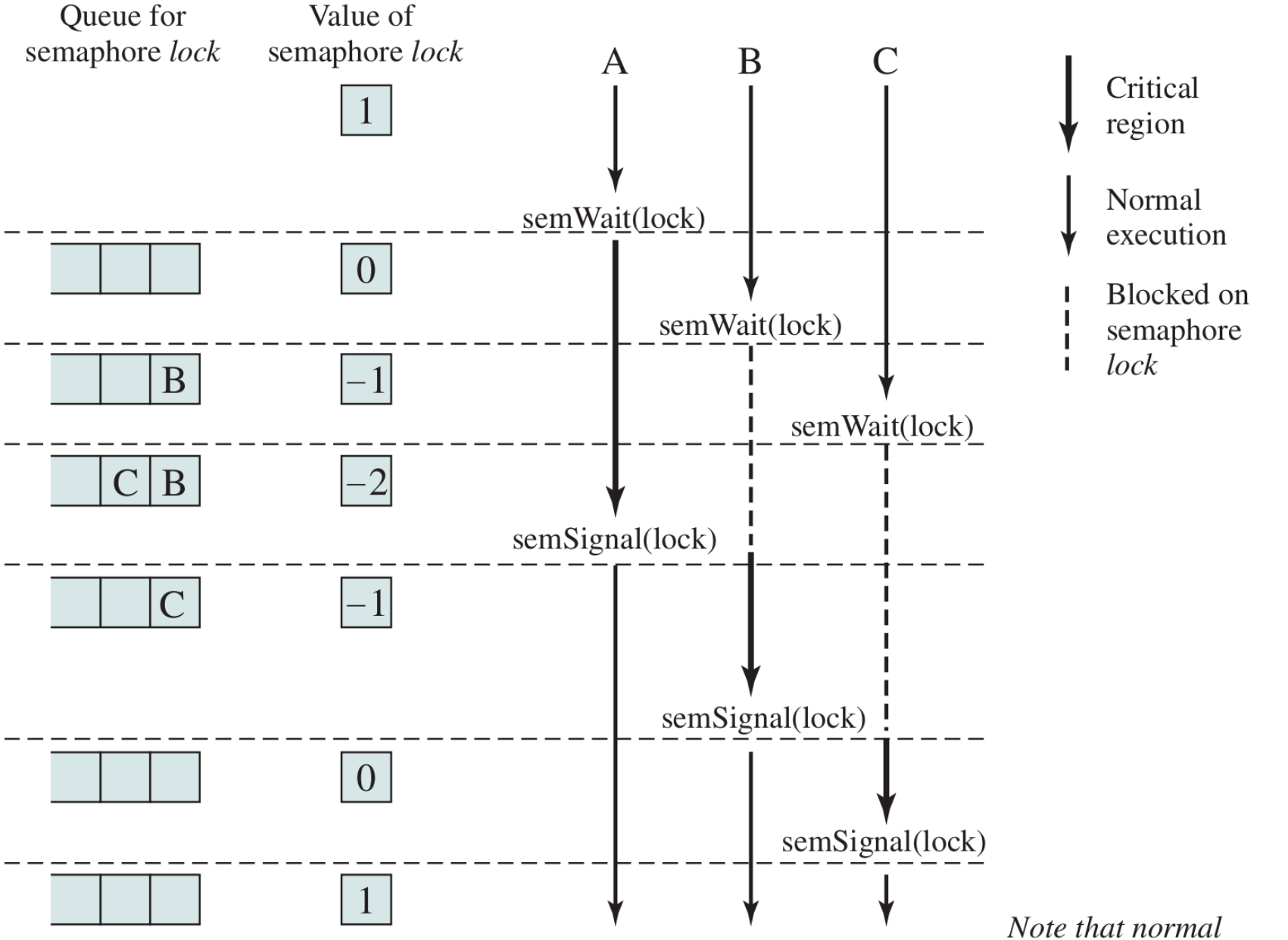
signal

```
1: procedure signal(S)
2:   var S: semaphore;
3:   begin
4:     S.value := S.value + 1;
5:     if S.value <= 0 then
6:       begin
7:         wakeup(S.L.first);
8:         pop(S.L);
9:       end
10:  end
```

用信号量实现进程同步

```
1:  var mutex semaphore := 1;
2:
3:  procedure P(i: integer):
4:  begin
5:    repeat
6:      wait(mutex);
7:      <critical section>;
8:      signal(mutex);
9:      <remainder section>;
10:   forever
11: end
12:
13: parbegin
14:   P(1); P(2); ...; P(n);
15: parend
```





Note that normal execution can proceed in parallel but that critical regions are serialized.

基于信号量的同步控制

```
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;

void P(int i) {
  while (true) {
    wait(s);
    /* critical section */;
    signal(s);
    /* remainder */;
  }
}
void main() {
  parbegin (P(1), P(2),..., P(n));
}
```

信号量正负的含义

- `s.value ≥ 0`: is the number of processes that can execute `wait(s)` without suspension.
- `s.value < 0`: the magnitude of s.value is the number of processes suspended in `s.queue`.

实例

- Unix/Linux
  - 锁(mutex/lock)，条件变量(cond)，信号量(sem)
- Windows
  - 事件(Event)
  - 锁(Mutex)
  - 信号量(Semaphore)
  - 临界区(CriticalSection)
- 不足
  - 配对使用
    - acquire/relea
    - lock/unlock
    - wait/signal

问题：
- 易读性差
- 难于维护
- 容易出错

并发Bug检测：
- 原子性破坏
- 顺序性破坏
- 死锁

AND型信号量

- 将进程在整个运行过程中需要的所有资源，一次性全都地分配给进程，待进程使用完后再一起释放。
  - 只要尚有一个资源未能分配给进程，其它所有可能为之分配的资源，也不分配给它
- 原子操作
  - 要么全部分配到进程，要么全不分配 .
- 在wait操作中，增加了一个“AND”条件，故称为“AND同步”，或称为“同时wait操作”.

原语原型

swait

```
1: procedure swait(S1, S2, ..., Sn)
2:   if S1 >= 1 and ... Sn >= 1 then
3:     for i := 1 to n do
4:       Si := Si - 1
5:     end
6:   else
7:     enqueue(current_process, Sj.L);
8: end
```

?

ssignal

信号量集

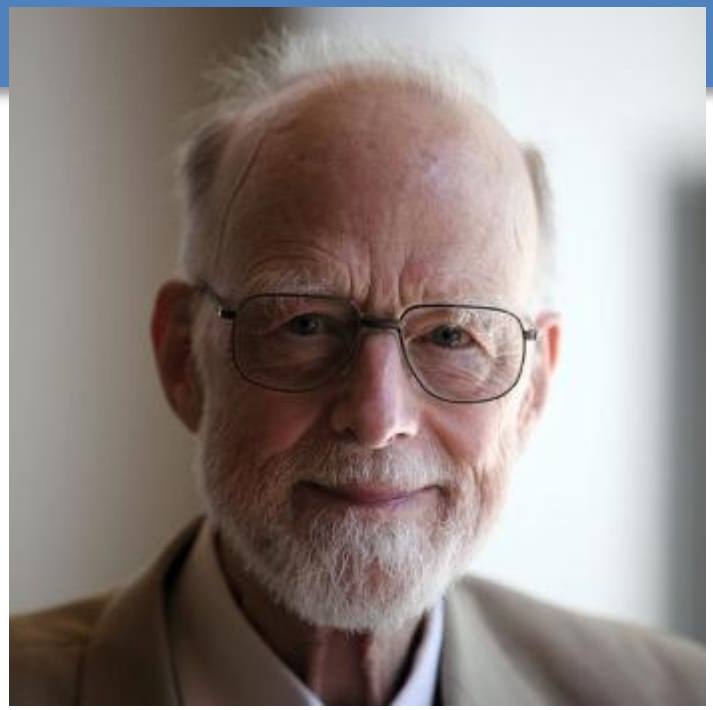
- 采用信号量集来控制，可以分配多个资源
  - swait(S1, t1, d1, …, Sn, tn, dn）
  - ti：资源下限； di：需求量

```
1: procedure swait(S1, t1, d1, ..., Sn, tn, dn)
2:   if S1 >= t1 and ... Sn >= tn then
3:     for i := 1 to n do
4:       Si := Si - dn
5:     end
6:   else
7:     enqueue(current_process, Sj.L);
8: end
```

管程——集中式进程同步

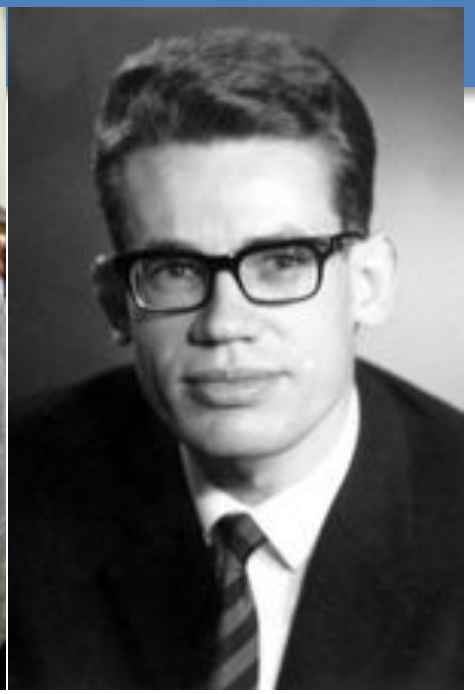
- 共享资源用共享数据结构表示，资源管理程序可用对该数据结构进行操作的一组过程来表示
- 这样一组相关的数据结构和过程称为——管程（Monitor）

- 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。
  - C. A. R. Hoare & Per Brinch Hansen



Charles Antony Richard Hoare
(11 January 1934)

- Turing Award @ 1980
- Quicksort @ 1960, 26
- Hoare Logic



Per Brinch Hansen
(November 13, 1938 – July 31, 2007)

- Concurrent Pascal @ 1975
- Operating System Principles @ 1973

管程的组成

- 局部于该管程的共享数据，这些数据表示了相应资源的状态；
- 针对上述数据的一组过程；
- 对局部于该管程的数据的初始化。

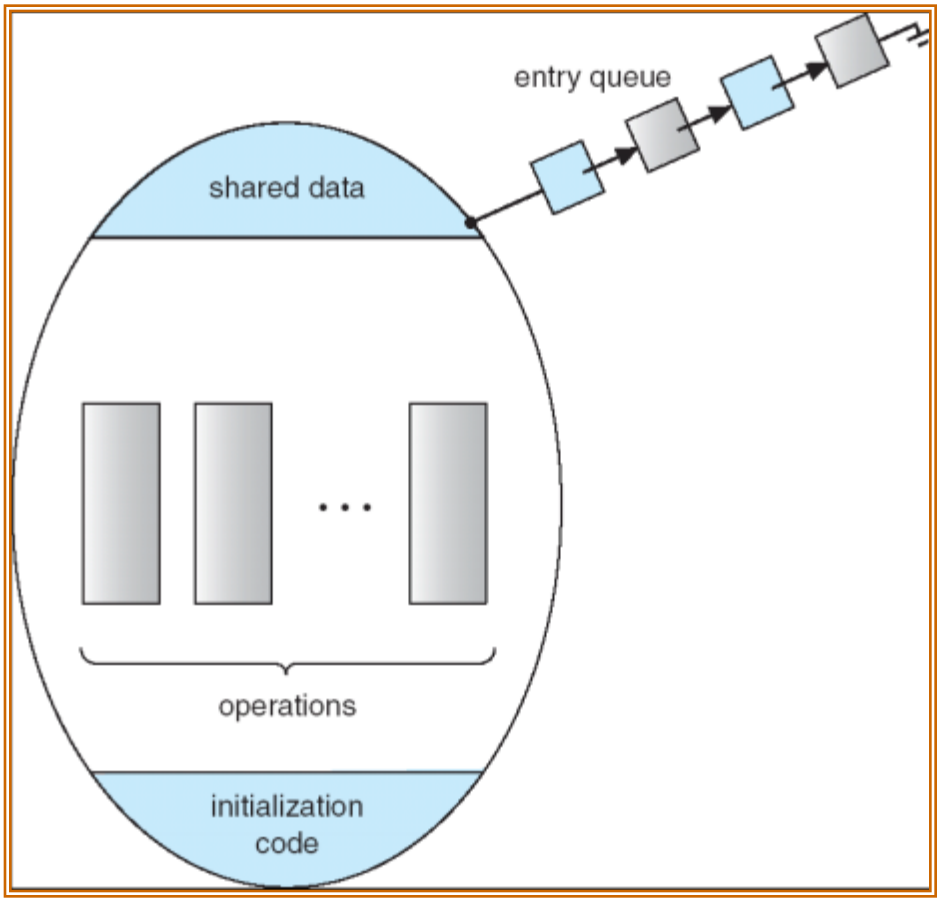
- 面向对象：封装（encapsulation）

管程的结构

```
monitor monitor-name
{
   shared variable declarations
   procedure P1 (…) { …. }
   …

   procedure Pn (…) {……}

   Initialization code
}
```



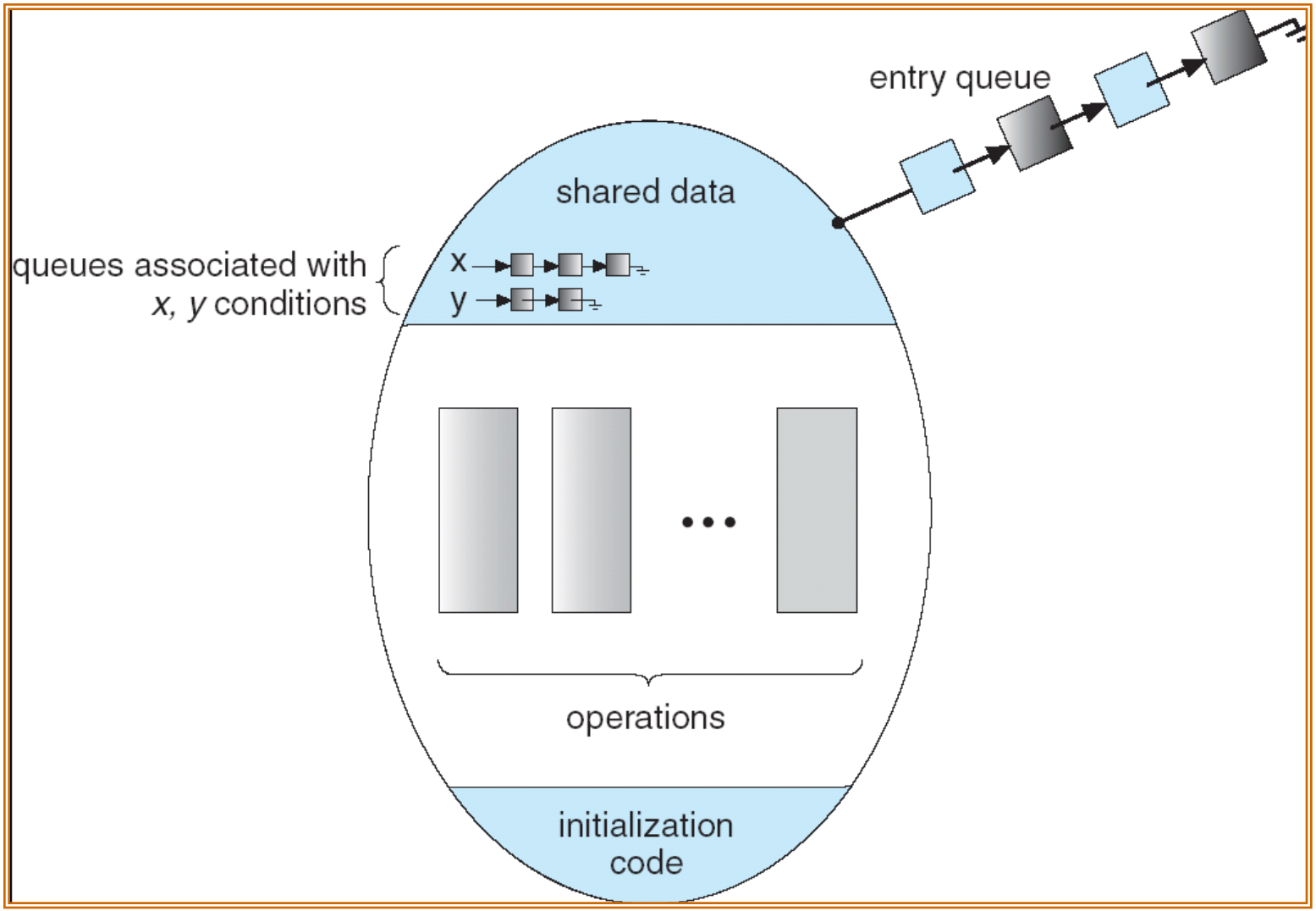
管程的特点

- 模块化（Modularization）
  - 管程是一个基本程序单位，可以单独编译;
- 抽象数据类型（Abstraction)
  - 管程中不仅有数据，而且有对数据的操作;
- 信息隐藏（Encapsulation）
  - 管程外可以调用管程内部定义的函数，但函数具体实现外部不可见;
  - 局部数据变量只能被管程的过程访问，任何外部过程都不能访问。

管程同步

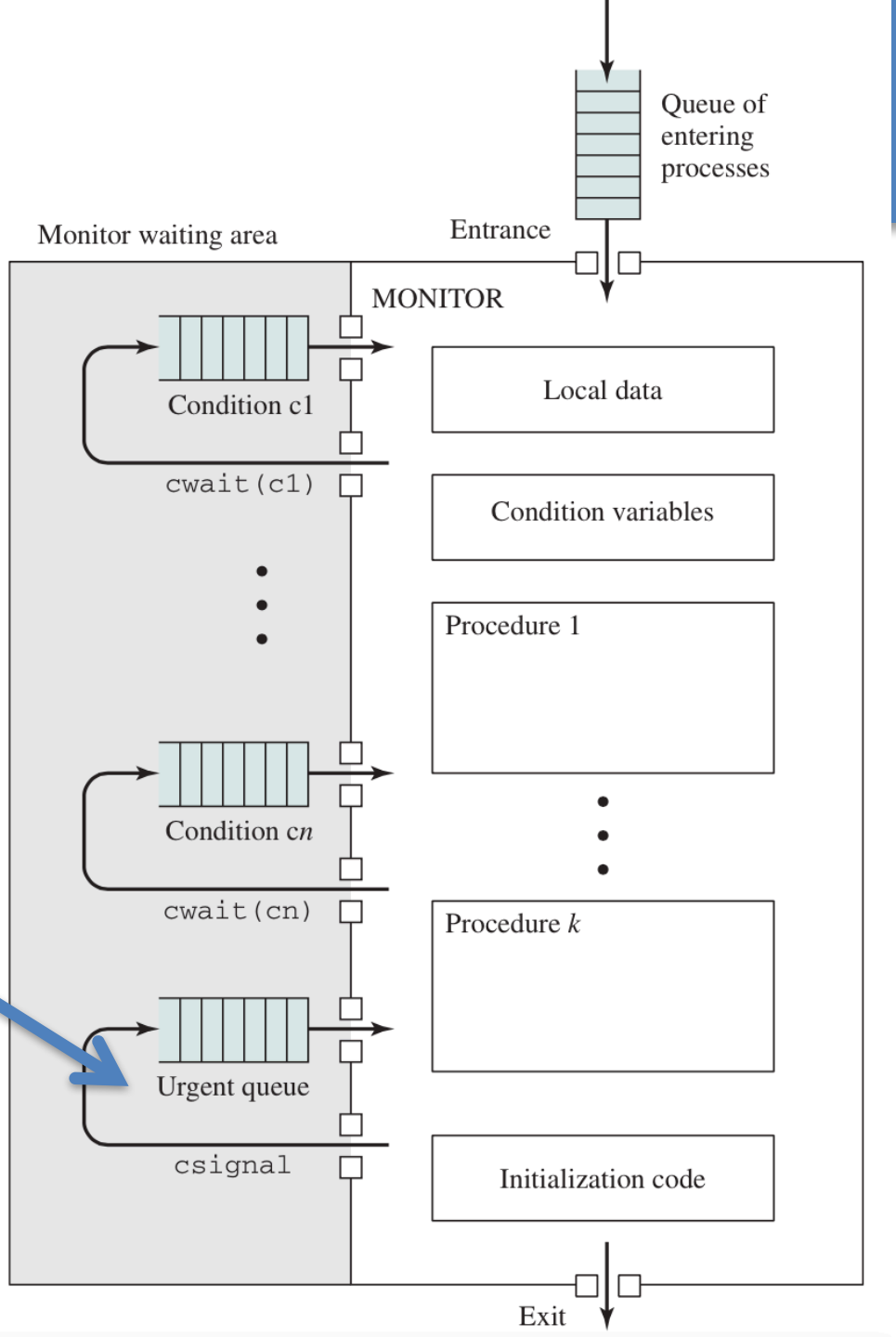
- 进程通过调用管程的一个过程进入管程；
- 在任何时候，只能有一个进程在管程中执行；调用管程的任何其他进程都被挂起，以等待管程变成可用。
- 条件变量提供同步支持。条件变量包含在管程中，并且只有在管程中才能被访问：
  - cwait(x)：调用进程的执行在条件x上挂起，管程现在可被另一进程使用。
  - csignal(x)：恢复阻塞在x上的进程。

管程的同步


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

条件变量

csignal() 之后Block



注意

- 条件变量不完全等同 “信号量”
- csignal(x)时，如果等待x的队列为空，则信号无任何作用（丢失），而信号量会保留其值
  - 对调度的要求：先调度x.queue

管程实例：Java的synchronized

```
public class SynchronizedCounter {
    private int c = 0;
    private ind d = 1;

    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return d;
    }
}
```

说明

- First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
- Second, when a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.
- http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

机理

- Synchronization is built around an internal entity known as the intrinsic lock or monitor lock.
- Every object has an intrinsic lock associated with it.

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

- http://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

Java中的锁

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

谢谢！