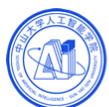


第5章 并发：互斥与同步

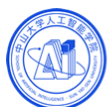
并发原理
互斥的软硬件实现
信号量
管程
消息传递
读者-写者问题



操作系统设计的核心问题

■ 进程与线程的管理（本章中进程代表线程）

- 多道程序设计技术——管理单[核]处理器系统中的多个进程
- 多处理技术——管理多[核]处理器系统中的多个进程
- 分布式处理技术——管理多台分布式计算机系统系统中的多个进程



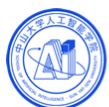
操作系统设计的核心问题

■ 并发（**concurrency**） 多道程序 / 多任务处理

- 所有问题的基础/根源
- 操作系统设计的基础

■ 并发发生的场合

- 多个应用程序（多道）
- 结构化应用程序（模块）
- 操作系统结构（结构化/模块）



并发相关术语

原子操作——不可分割

- 临界区（**critical section**）——不允许多个进程同时进入的一段访问共享资源的代码
- 死锁（**deadlock**）——两个及以上进程，因每个进程都在等待其他进程操作（如释放资源）而不能继续执行
- 互斥（**mutual exclusion**）——一个进程在临界区访问共享资源时，不允许其他进程进入访问
- 竞态（**race condition**）——多个进程/线程读写共享数据，其结果依赖于它们执行的相对速度/相对时序
- 饥饿（**starvation**）——可运行进程长期未被调度执行

并发性

■ 并发性引发的问题

- 资源竞争、共享、分配管理困难
- 难调试（程序执行结果不可再现）

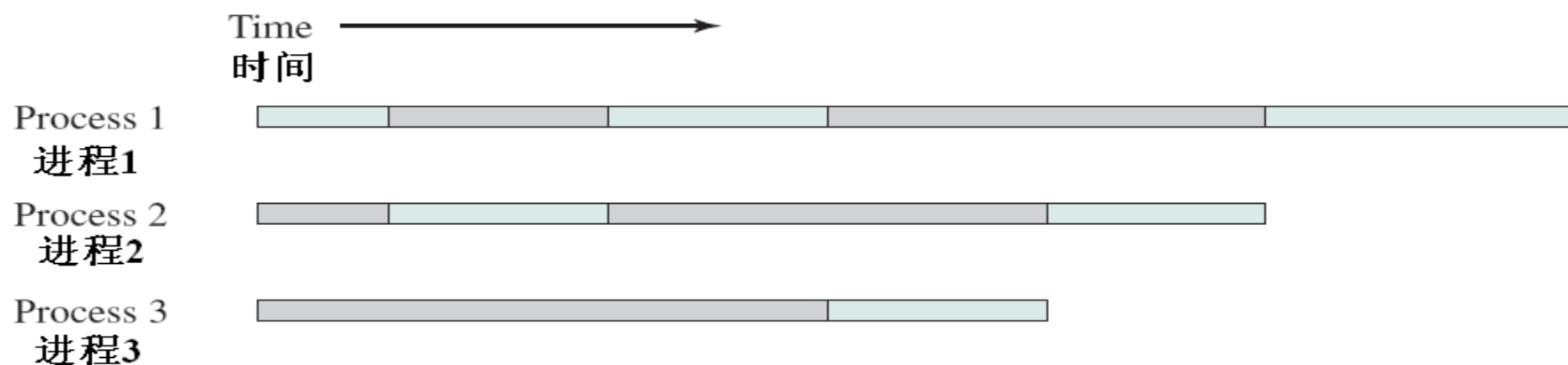
■ 进程的互斥机制

- 软件方法（**Dekker**算法、**Peterson**算法）
- 硬件件方法（关中断、专用指令**TestSet/Exchange**）

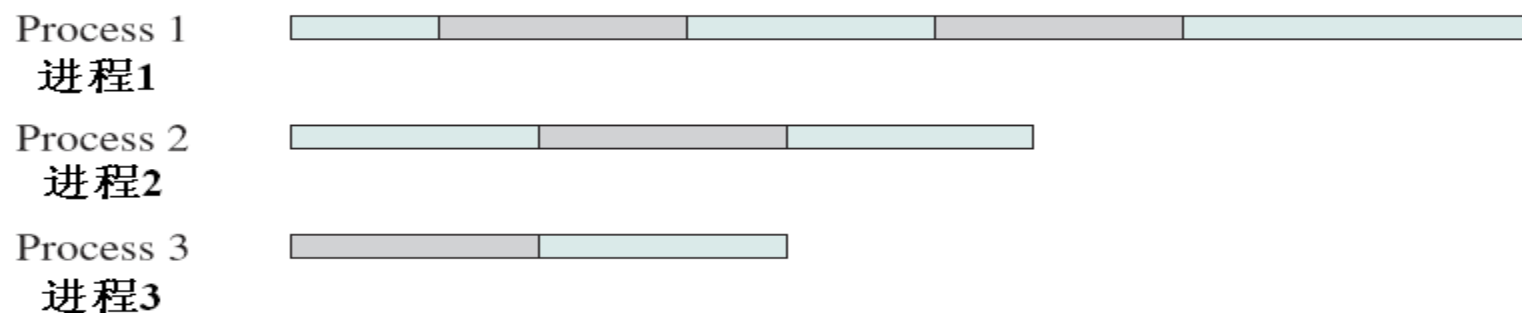
■ 进程的相互作用

- 通过共享的竞争
- 通过共享的合作
- 通过通信的合作

5.1 并发的原理



(a) Interleaving (multiprogramming; one processor)
交错（多道程序设计，单处理器）



(b) Interleaving and overlapping (multiprocessing; two processors)
交错与重叠（多道程序设计，双处理器）

Blocked
阻塞

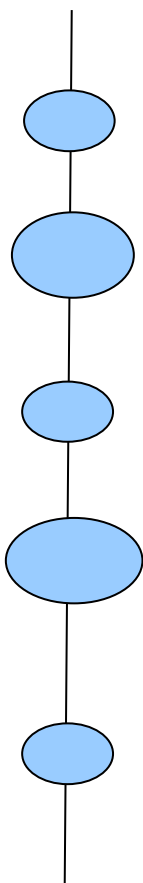
Running
运行

Multiprogramming and Multiprocessing
多道程序设计与多处理

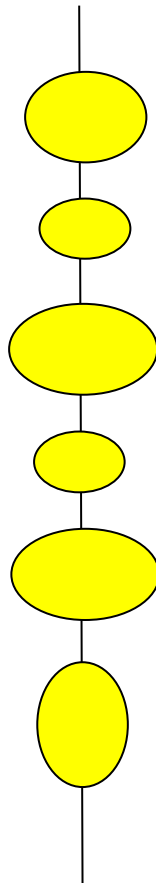


珍珠串问题

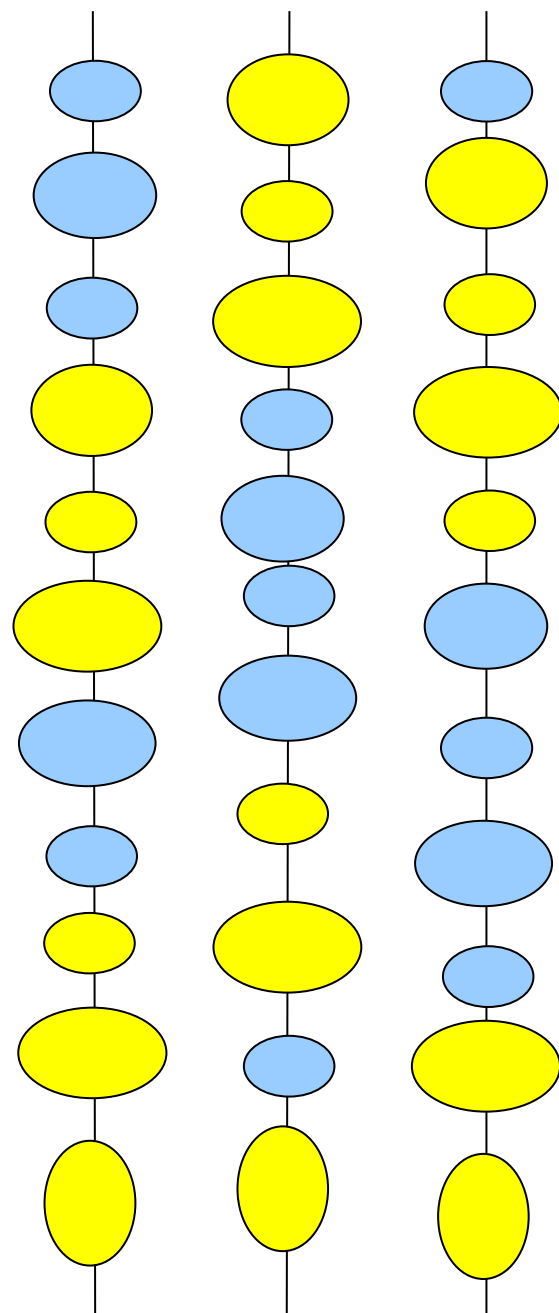
- 2串用线穿着的珠宝，各5和6个珍珠。
- 将两串珍珠从任意位置剪开，再拼接成一个珍珠串，要求保持原串中的相对顺序。
- 可能穿成多少种不同的珍珠串？



珍珠串1



珍珠串2



l: 珠宝串数

n: 每串的珠宝数

可能穿成: $(n \cdot l)! / (l \cdot (n!))$

并发的基本特征

■ 并发

在同一时间段内活动的进程或线程，在此期间，它们可能交替地共享相同的资源

■ 异步性

- 相对执行速度不可预测
- 多道程序系统的基本特性

并发的基本特征

■ 影响进程执行速度的因素

- 其他进程的活动
- 操作系统处理中断的方式
- 操作系统的调度策略

■ 问题

- 全局资源的共享充满危险
- 操作系统对资源分配的管理难以达到最优
- 调试程序设计错误非常困难（不可再现性）

5.1.1 并发产生的错误

售票程序

ticksale:

•

mov ax,tickqty;

dec ax;

mov tickqty,ax;

•

}

售票点P1和P2, 余票 tickyqty=100

Process P1

•

mov ax,tickqty;

•

•

•

dec ax;

mov tickqty,ax;

Process P2

•

•

mov ax,tickqty;

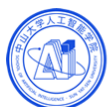
dec ax;

mov tickqty,ax;

•

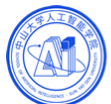
•

控制关键代码片段的并发时序!



5.1.2 竞态 (race condition)

- 在并发环境中发生
- 多个进程共享数据
- 多个进程读取且至少一个进程写入
- 共享数据产生**错误结果**，具体结果取决于进程执行的相对速度

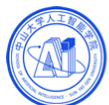


5.1.3 操作系统必须考虑的问题

- 并发环境中**跟踪**每个进程，知道它们的状态
- 为每个进程**分配**和**回收**各种资源
 - 处理机
 - 存储器
 - 文件
 - I/O设备
- **保护**进程拥有的数据和物理资源

5.1.3 操作系统必须考虑的问题

- 防止并发竞态发生，保证进程的结果正确，与相对执行速度无关
 - 内核并发发生竞态
 - 用户进程并发竞态



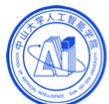
5.1.4 进程间的相互作用

■ 间接作用

- 因为共享而竞争
- 通过共享实现合作

■ 直接作用

- 通过通信的合作



进程间的竞争现象

■ 特点:

- 独立设计的进程，每个进程不知道其他进程的存在（“萍水相逢”、“我不知道你是谁”）
- 两个或更多进程在各自的执行过程中需要访问相同的资源（I/O设备、存储器、**CPU**、时钟等）（“独木桥上，狭路相逢”）
- 进程之间没有信息交换的要求（“各走各的路，井水不犯河水”）

进程间的竞争现象

■ 相互间产生的影响：

- 执行结果不会受影响
- 执行时间受影响

■ 竞争引发的控制问题

- 互斥 (mutual exclusion)
- 死锁 (deadlock)
- 饥饿 (starvation)

互斥的关联概念

■ 互斥 (mutual exclusion)

- 多个进程需要访问一个共享的资源时，任何时候只能有一个访问这个资源

■ 临界资源 (critical resource)

- 不可操作使用的资源

■ 临界区 (critical section)

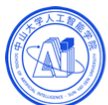
- 访问临界资源的那部分代码

■ 死锁 (deadlock)

- 一组进程中，每个进程都无限等待该组进程中另一进程所占用的资源

■ 饥饿 (starvation)

- 一组进程中，某个或某些进程无限等待该组进程中其他进程所占用的资源



进程间通过共享的合作

■ 特点

- 没有意识到其他进程的存在，需要维护数据的完整性
- 共享变量、文件或数据库等

■ 相互间产生的影响：

- 执行结果可能会受影响
- 执行时间受影响

P1:

•

$a = a + 1;$

•

$b = b + 1;$

•

P2:

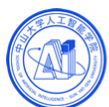
•

•

$b = 2 * b;$

•

$a = 2 * a;$

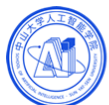


进程间通过共享的合作

■ 共享引发的控制问题

- 互斥
- 死锁
- 饥饿
- 数据的一致性

| P1: | P2: |
|--------------|--------------|
| • | • |
| $a = a + 1;$ | • |
| • | $b = 2 * b;$ |
| $b = b + 1;$ | • |
| • | $a = 2 * a;$ |



进程间通过通信的合作

■ 特点

- 进程直接知道合作伙伴
- 采用消息传送的方式通信（发送/接收消息）

■ 相互间产生的影响

- 执行结果可能会受影响
- 执行时间受影响

■ 引发的控制问题:

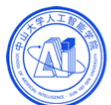
- 死锁
- 饥饿

5.1.5 解决互斥问题的要求

- 进程并发，完全自由无约束，可能产生竞态
- 互斥是并发中防止产生错误的一种模式
- 操作系统解决互斥的机制/方案，要满足一定的条件

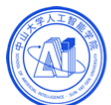
5.1.5 解决互斥问题的条件

- 具有相同资源或共享对象的临界区的所有进程中，一次只允许一个进程进入临界区（**强制排它**）
- 在非临界区停止的进程不干涉其他进程（**充分并发**）
- 没有进程在临界区中时，任何需要访问临界区的进程必须能够立即进入（**空闲让进**）
- 不允许出现一个需要访问临界区的进程被无限延迟（**有限等待**）



5.1.5 解决互斥问题的条件

- 相关进程的执行速度和处理机数目没有任何要求或限制（**满足异步**）
- 当进程不能进入临界区，应该立即释放处理机，防止进程忙等待（**让权等待**）



互斥机制应用框架

```
void P1() {  
    ... /* preceding code */  
    enter_cs(Ra);  
    ... /* cs Ra */  
    exit_cs(Ra);  
    ... /* following code */  
}
```

```
void P2() {  
    ....  
}
```

```
void Pn() {  
    ....  
}
```

N个进程并发，竞争资源Ra

每个进程都会有一段代码操作Ra

Cs: 临界区

为了确保结果正确，必须互斥操作Ra

实现互斥的方法

■ 软件方法

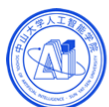
- Dekker算法
- Peterson算法

■ 硬件方法

- TestSet指令
- Exchange指令

■ 操作系统或程序设计语言的支持

- 信号量
- 管程
- 消息机制

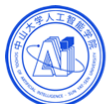


第一种尝试

用 turn 标记位轮转

```
/* PROCESS 0 */           /* PROCESS 1 */  
.  
.  
while (turn != 0)  
    /* do nothing */ ;  
/* critical section */  
turn = 1;  
.  
.  
while (turn != 1)  
    /* do nothing */;  
/* critical section */  
turn = 0;  
.
```

(a) First attempt



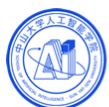
第一种尝试的特点

- 可以保证互斥
- 硬性规定进入的顺序
 - 两个进程轮流进入临界区

■ 存在问题

分析：难以满足并发需求！

- 忙等待 (busy waiting)：为了等待一事件的发生，重复执行一段循环代码 ---- 白白消耗CPU时间
- 必须轮流进入临界区 ---- 不合理，限制推进速度
- 如果一个进程失败，另一个将被永远阻塞

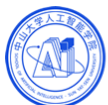


第二种尝试

用 `flag[i]` 标志进程 `i` 进入临界区

```
/* PROCESS 0 */      /* PROCESS 1 */
.
.
while (flag[1])
    /* do nothing */;
flag[0] = true;
/*critical section */
flag[0] = false;
.
.
while (flag[0])
    /* do nothing */;
flag[1] = true;
/* critical section */
flag[1] = false;
.
```

(b) Second attempt



第二种尝试的特点

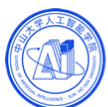
■ 每个进程应有自己进入临界区的“钥匙”

- 进程设置标志表示自己进入和离开
- 可以检查对方标志
- 先查对方标志再设置自己进入临界区的标志

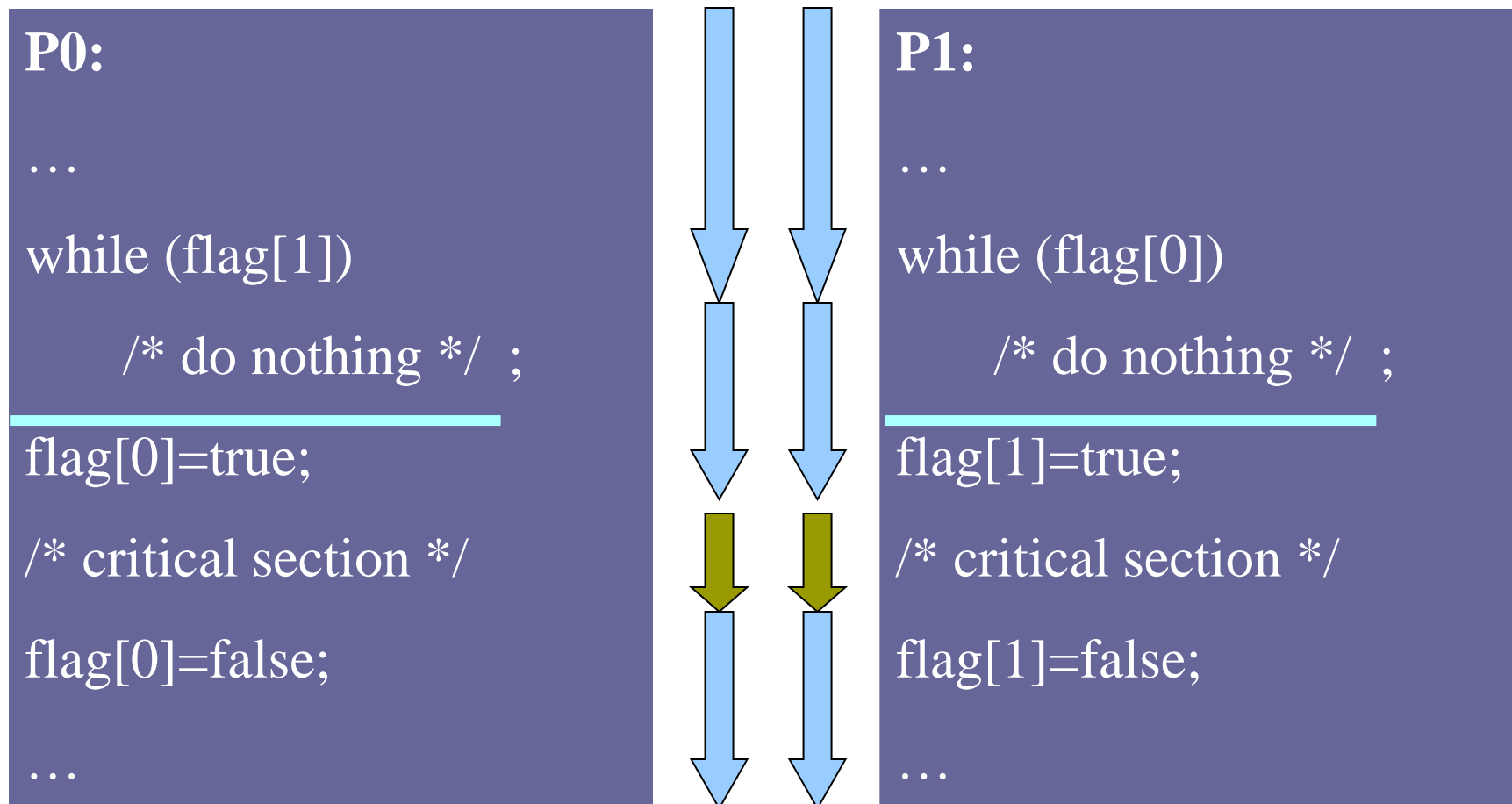
■ 存在问题

分析：错误方案，不能保证进程的运行
结果与执行速度无关

- 一个进程在临界区内失败，另一进程永远被阻塞
- 不能保证互斥！（见示意图）

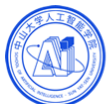


第二种尝试的失效示意图



如果在横线处被切换，

两个进程可能同时进入临界区！

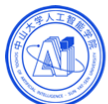


第三种尝试

将 `flag[i]` 标志的设置，提前到循环等待之前

```
/* PROCESS 0 */          /* PROCESS 1 */  
  
.  
.  
flag[0] = true;  
while (flag[1])  
    /* do nothing */;  
/* critical section */  
flag[0] = false;  
.  
  
.  
.  
flag[1] = true;  
while (flag[0])  
    /* do nothing */;  
/* critical section */  
flag[1] = false;  
.
```

(c) Third attempt



第三种尝试的特点

先表示自己想进入临界区，再检查对方是否已进入

- 可以保证互斥

- 问题

- 可能导致死锁!

- 死锁产生的原因

- 两进程都坚持要进入

第三种尝试的失效示意图

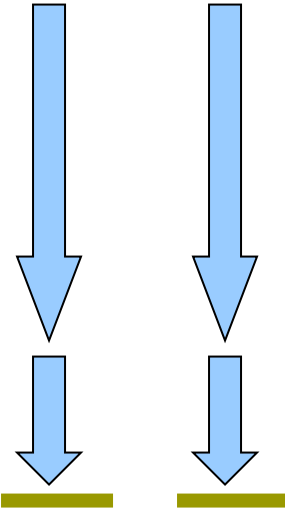
P0:

...

flag[0]=true;

while (flag[1]);

...



P1:

...

flag[1]=true;

while (flag[0]);

...

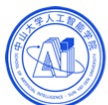
第四种尝试

在循环等待中用延时给其他进程进入的机会

```
/* PROCESS 0 */
.
.
flag[0] = true;
while (flag[1]) {
    flag[0] = false;
    /* delay */
    flag[0] = true;
}
/* critical section */
flag[0] = false;
.

/* PROCESS 1 */
.
.
flag[1] = true;
while (flag[0]) {
    flag[1] = false;
    /* delay */
    flag[1] = true;
}
/* critical section */
flag[1] = false;
```

(d) Fourth attempt



第四种尝试的特点

■ 解决思路

- 礼让，等一会

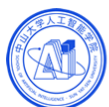
■ 可以保证互斥

■ 死锁与活锁

- 死锁: 都想进入临界区，但均不能进入
- 活锁: 本来可以进入临界区，但均不能进入

■ 问题

- 会导致活锁



第四种尝试的活锁时序

P0:

...

```
flag[0]=true;
```

```
while (flag[1]) {
```

```
    flag[0]=false;
```

```
    delay();
```

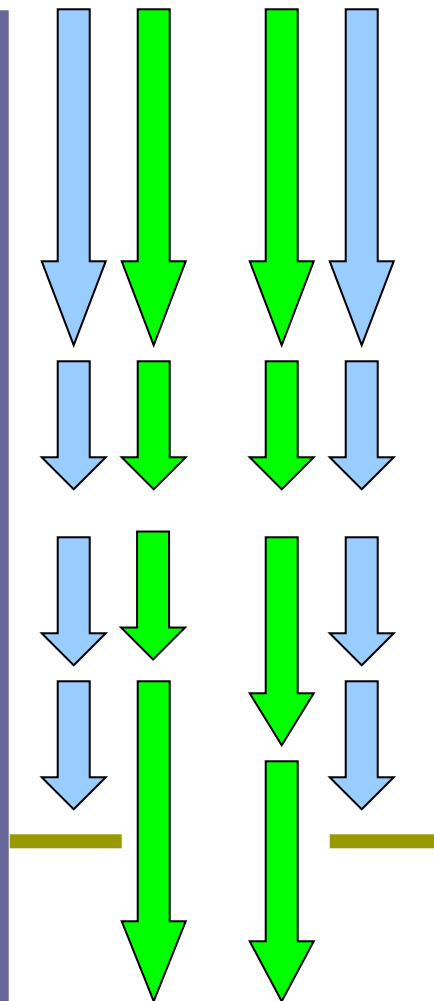
```
    flag[0]=true;
```

```
}
```

```
/* critical section */
```

```
flag[0]=false;
```

...



P1:

...

```
flag[1]=true;
```

```
while (flag[0]) {
```

```
    flag[1]=false;
```

```
    delay();
```

```
    flag[1]=true;
```

```
}
```

```
/* critical section */
```

```
flag[1]=false;
```

...

Dekker 算法

- 1965年荷兰数学家T. J. Dekker
- 避免“无原则”的礼让
- 规定各进程进入临界区的进入顺序
- 全局数组变量 **flag** 表示进入临界区的“意愿”
- 全局变量 **turn** 解决进入顺序



Dekker 算法

```
boolean flag[2]; int turn;
```

```
void P0() {
```

```
    while (true) {
```

```
        flag[0] = true; // 自己想进临界区
```

```
        while (flag[1]) {
```

```
            if (turn == 1) {
```

```
                flag[0] = false;
```

```
                while (turn == 1)
```

```
                    flag[0] = true;
```

```
            }
```

```
        }
```

```
        /* critical section */
```

```
        turn = 1;
```

```
        flag[0] = false;
```

```
        /* remainder */
```

```
    }
```

```
}
```

```
// flag[1]==false 时进入临界区
```

```
// flag[1]==true时等待
```

```
// turn==1时礼让
```

```
/* do nothing */;
```

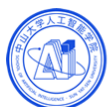
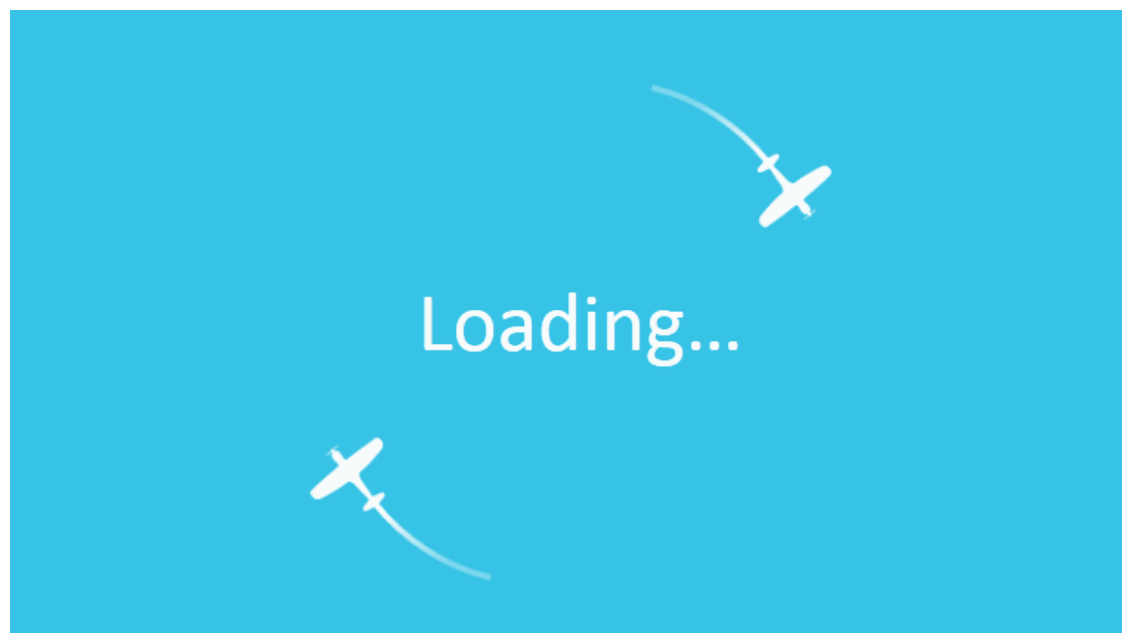
Dekker 算法 (续)

```
void P1() {  
    while (true) {  
        flag[1] = true; // 自己想进临界区  
        while (flag[0]) {  
            if (turn == 0) {  
                flag[1] = false;  
                while (turn == 0) /* do nothing */;  
                flag[1] = true;  
            }  
        }  
        /* critical section */  
        turn = 0;  
        flag[1] = false;  
        /* remainder */  
    }  
}  
  
// flag[0]==false时进入临界区  
// flag[0]==true时等待  
// turn==0时礼让  
  
void main () {  
    flag[0] = false;  
    flag[1] = false;  
    turn = 1;  
    parbegin (P0, P1);  
}
```

Dekker 算法 (续)

■ 算法的问题

- 逻辑复杂
- 正确性难证明
- 存在轮流问题
- 存在忙等待



Peterson 算法

- 1981年数学家G. L. Peterson
- 简单出色（不存在轮流问题）
- flag 和turn的含义与Dekker相同

- 先设 $\text{turn} = \text{别人}$ ，只有“ $\text{flag}[\text{别人}]$ ”和“ $\text{turn} = \text{别人}$ ”同时为真时才循环等待
- 参见附录A“并发主题”

Peterson 算法 (续)

```
boolean flag [2]; int turn;
```

```
void P0() {
```

```
    while (true) {
```

```
        flag [0] = true;
```

```
        turn = 1;
```

```
        while (flag [1] && turn == 1)
```

```
            /* do nothing */;
```

```
        /* critical section */
```

```
        flag [0] = false;
```

```
        /* remainder */
```

```
    }
```

```
}
```

```
void P1() {
```

```
    while (true) {
```

```
        flag [1] = true;
```

```
        turn = 0;
```

```
        while (flag [0] && turn == 0)
```

```
            /* do nothing */;
```

```
        /* critical section */
```

```
        flag [1] = false;
```

```
        /* remainder */
```

```
    }
```

```
}
```

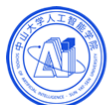
```
void main() {
```

```
    flag [0] = false;
```

```
    flag [1] = false;
```

```
    parbegin (P0, P1);
```

```
}
```



硬件实现方法——中断禁用

中断禁用指令

临界区

中断启用指令

■ 中断禁用（关中断）原理

■ 单CPU体系结构

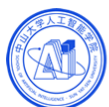
- 如果进程访问临界资源时（执行临界区代码）不被中断，就能保证互斥地访问

限制了处理器交替执行各进程的能力

■ 途径

不能用于多[核]处理器结构

- 使用关/开中断指令
- x86的开/关指令为STI/CLI



硬件实现方法——专用指令

■ 适用范围

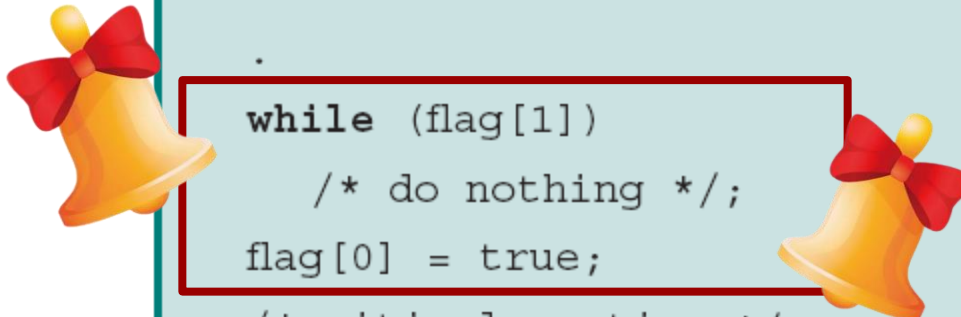
- 单处理器或共享主存多[核]处理器结构
- 对同一存储单元的访问是互斥的

■ 软件算法第二种尝试失败的原因

- 检测 `flag[1]` 和置位 `flag[0]` 在一个指令周期完成不会出错

第二种尝试

用 `flag[i]` 标志进程 `i` 进入临界区



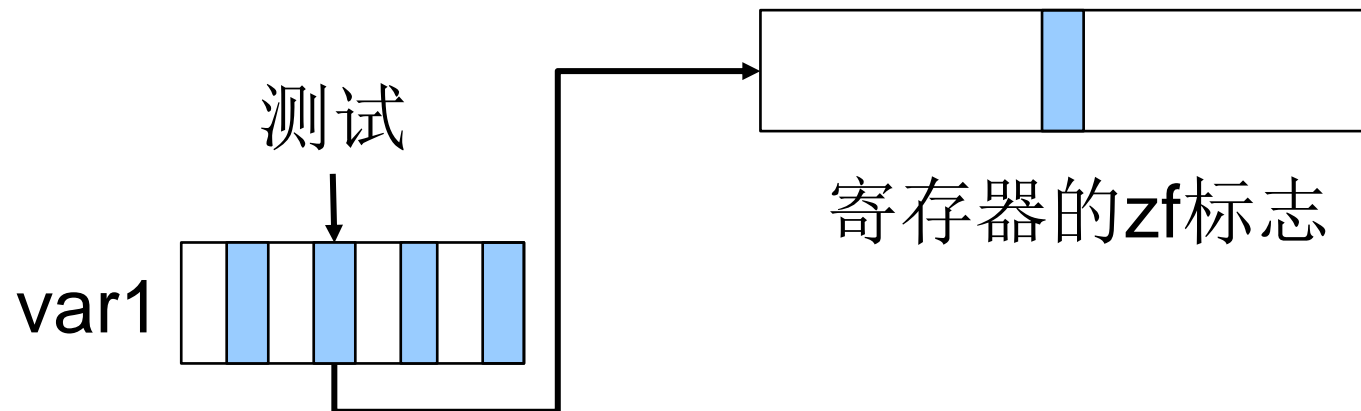
```
/* PROCESS 0 */      /* PROCESS 1 */  
.  
.  
while (flag[1])  
    /* do nothing */;  
flag[0] = true;  
/*critical section */  
flag[0] = false;  
.  
.  
while (flag[0])  
    /* do nothing */;  
flag[1] = true;  
/* critical section */  
flag[1] = false;  
.
```

(b) Second attempt

加入金钟罩：保证在一个指令周期完成

TestSet指令(TS)

- 定义（逻辑）——比较并交换指令的bool特例
- 原子指令
- 指令功能



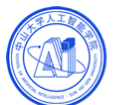
测试内存变量 **var1** 某一位的值：

- 0：设置标志寄存器的zf标志置位(1)，而且var1这位变1
- 1：设置标志寄存器的zf标志复位(0)，而且var1这位变1

TestSet指令 – 解决方案

- 利用TestSet实现锁机制
- 锁机制自旋锁(spin lock)
 - 内核自用的一种互斥机制
 - 一个锁变量(其实是一个二进制位) lockbit
 - 加锁操作lock(lockvar)
 - 解锁操作unlock(lockvar)

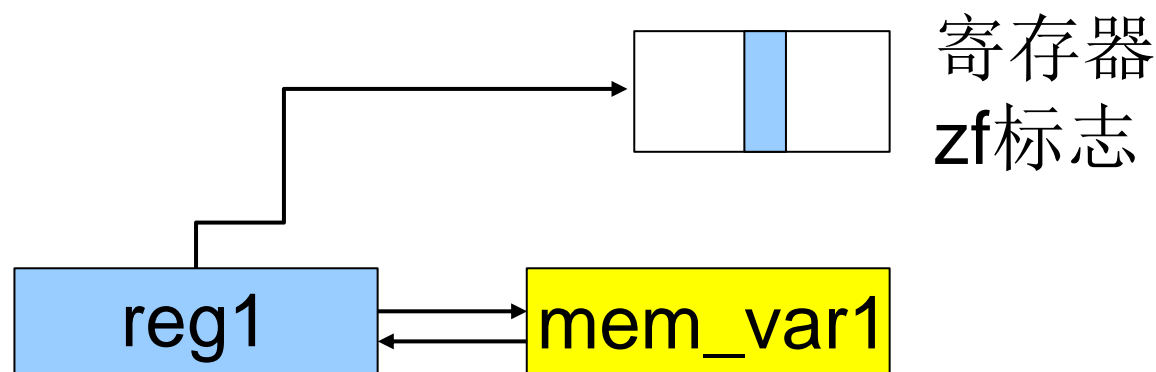
```
Lockvar db 0
proc lock(lockvar)
Loop: testset lockvar
      jnz loop
      ret
end lock
proc unlock(lockvar)
      mov lockvar, 0
      ret
end unlock
```



exchange 指令

- 定义：交换一个寄存器和内存单元的内容

- 原子操作



- x86 CPU的对应指令为XCHG

- 指令功能

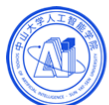
根据寄存器reg1与内存单元var1的值对换，并根据结果设置标志寄存器的z标志。

exchange 指令

- 利用 **xchg** 实现锁机制
- 锁机制
 - 内核自用的一种互斥机制，又称为自旋锁(spin lock)
 - 一个锁变量 **lockvar**
 - 加锁操作 **lock(lockvar)**
 - 解锁操作 **unlock(lockvar)**

```
Lockvar db 0
```

```
proc lock(lockvar)  
    mov ax, 0FFh  
Loop: xchg ax, lockvar  
    jnz loop  
    ret  
end lock  
proc unlock(lockvar)  
    mov lockvar, 0  
    ret  
end unlock
```



exchange 指令

■ 互斥方法

- 设置锁变量lockvar，初值为0
- 临界区前，lock(lockvar)
- 临界区后，unlock(lockvar)

```
Lockvar db 0
```

```
proc lock(lockvar)  
    mov ax, 0FFh  
Loop: xchg ax, lockvar  
    jnz loop  
    ret  
end lock  
proc unlock(lockvar)  
    mov lockvar, 0  
    ret  
end unlock
```

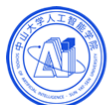
机器指令方法的优缺点

■ 优点

- 适用于单处理器或共享主存多[核]处理器系统, 进程数目任意
- 简单且易于证明
- 可以使用多个变量支持多个临界区

■ 缺点

- 忙等待 (busy waiting) / 自旋等待 (spin waiting)
- 可能饥饿
- 可能死锁



常用并发机制

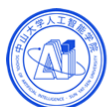
| 并发机制 | 说明 |
|-------|--|
| 信号量 | 用于进程间传递信号的一个整数值，只有初始化、增、减三种原子操作，可阻塞/解除阻塞进程 |
| 二元信号量 | 取值只为0和1的信号量 |
| 互斥量 | 似二元信号量，但要求为其加锁和解锁的须是同一进程 |
| 条件变量 | 一种数据类型，用于阻塞进程/线程，直到特定条件为真 |
| 管程 | 一种编程语言结构，封装了代表临界区的若干过程 |
| 事件标志 | 用于同步机制的内存字，其每个位关联不同的事件 |
| 信箱/消息 | 进程间交换信息的一种方法，也可用于同步 |
| 自旋锁 | 一种互斥机制，进程在一无条件循环中执行，等待锁变量值变为可用 |

5.3 信号量

红灯停、绿灯行

■ semaphore ['seməfo:(r)]

信号量、旗语、（铁
道）臂板信号装置



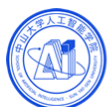
5.3 信号量



Edsger W. Dijkstra
(1930-2002)

■ 信号量机制

- 荷兰计算机科学家 E. W. Dijkstra 提出 (1965)，解决并发进程问题的第一个重要进展，**需操作系统支持**
- 两个或多个进程可以通过传递**信号**进行合作，从而可以迫使进程在指定位置暂停，直到它收到特定信号
- 信号量机制可以满足任何复杂的合作要求
- 信号量值用来表示可用资源的数目（非负整数）



5.3 信号量



Edsger W. Dijkstra
(1930-2002)

■ 信号量机制的组成

- 操作系统提供的用于进程并发控制的特殊数据结构
- 含有一个非负整数变量和三个专门操作

① 初始化

② P（荷兰语proberen，测试/通过）操作

③ V（荷兰语verhogen，增量/释放）操作

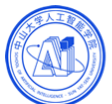
对信号量的操作

■ 初始化

- 通常将信号量的值初始化为**非负整数** (=可用资源数)

■ P操作/semWait操作/Down操作

- 信号量值减1
- 若信号量的值变成负数，则请求执行P操作的进程被**阻塞**



对信号量的操作

■ V操作/semSignal操作/Up操作

- 信号量的值加1
- 如果信号量的值不是正数（其绝对值=现被阻塞的进程数[等待队列的长度]），则使一个因执行P操作被阻塞的进程**解除阻塞**（唤醒）

无其他检查和修改信号量值的操作！



信号量和p、v原语的描述

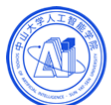
```
void p(semaphore s)
{
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */
        /* block this process */
    }
}
```

```
struct semaphore {
    int count;
    queueType queue;
};
```

```
void v(semaphore s)
{
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */
        /* place process P on ready list */
    }
}
```

二元信号量 (binary semaphore)

- 信号量的取值只能是0或1
- 和一般信号量具有相同的表达能力
- 因count不能小于0，需其他方法判断等待队列是否为空



二元信号量和操作 原语的描述

```
struct semaphore{  
    bool available;  
    queueType queue;  
} s;
```

```
void v(semaphore s) {  
    if (s.queue is empty) s.available = TRUE;  
    else //remove a process from s.queue and ready  
}
```

```
void p(semaphore s) {  
    if (s.available) s.available = FALSE;  
    else //place this process in s.queue and block  
}
```

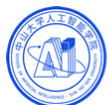
信号量的实际含义

■ 信号量 s ($s.count$) 的初值

- 系统中某类资源的数目，应该 ≥ 0

■ 进程执行 $P(s)$ / $\text{semWait}(s)$ 操作

- 申请一个单位的资源 ($s.count--$)
- 若 $s.count < 0$ ，资源已分配完毕，进程被操作系统阻塞在 s 的队列上----让权等待



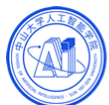
信号量的实际含义

■ 进程执行V(s)/semSignal(s)操作

- 释放一个单位资源 (s.count++)
- 若s.count ≤ 0 , 则唤醒一个等待进程

■ s.count

- ≥ 0 : 可用的资源数/可以执行P(s)而不会阻塞的进程数
- < 0 : |s.count|为在队列中等待的进程数



信号量的实现

■ 基本要求

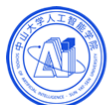
- 保证 **P** 和 **V**操作的原子性，实现信号量操作的互斥

■ 软件方案

- Dekker算法
- Peterson算法

■ 硬件支持方案

- 可以采用**TS**指令
- 关中断



信号量的TestSet指令实现

```
void p(semaphore s) {
```

```
    lock(s.flag);
```

```
    s.count--;
```

```
    if (s.count<0) Block(CurruntProcess, s.queue);
```

```
    unlock(s.flag); }
```

```
void v(semaphore s) {
```

```
    lock(s.flag);
```

```
    s.count++;
```

```
    if (s.count<=0) WakeUp(s.queue);
```

```
    unlock(s.flag); }
```

```
struct semaphore{
```

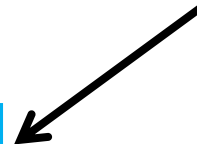
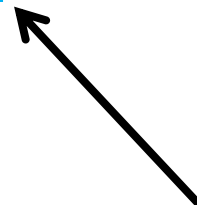
```
    lockbit flag;
```

```
    int count;
```

```
    queueType *queue;
```

```
} s;
```

临界区



信号量的关中断实现

```
void p(semaphore s) {
```

```
    inhibit interrupts;
```

```
    s.count--;
```

```
    if (s.count < 0) Block(CurruntProcess, s.queue);
```

```
    allow interrupts;}
```

```
void v(semaphore s) {
```

```
    inhibit interrupts;
```

```
    s.count++;
```

```
    if (s.count <= 0) WakeUp(s.queue);
```

```
    allow interrupts;}
```

```
struct semaphore{  
    int count;  
    queueType *queue;  
} s;
```

临界区

信号量的优缺点

■ 优点

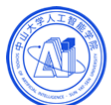
- 简单，而且表达能力强
- 用P、V操作可解决多种类型的同步/互斥问题

■ 缺点

- 不够安全，P、V操作使用不当可能产生死锁
- 遇到复杂同步互斥问题时实现复杂

信号量机制的应用

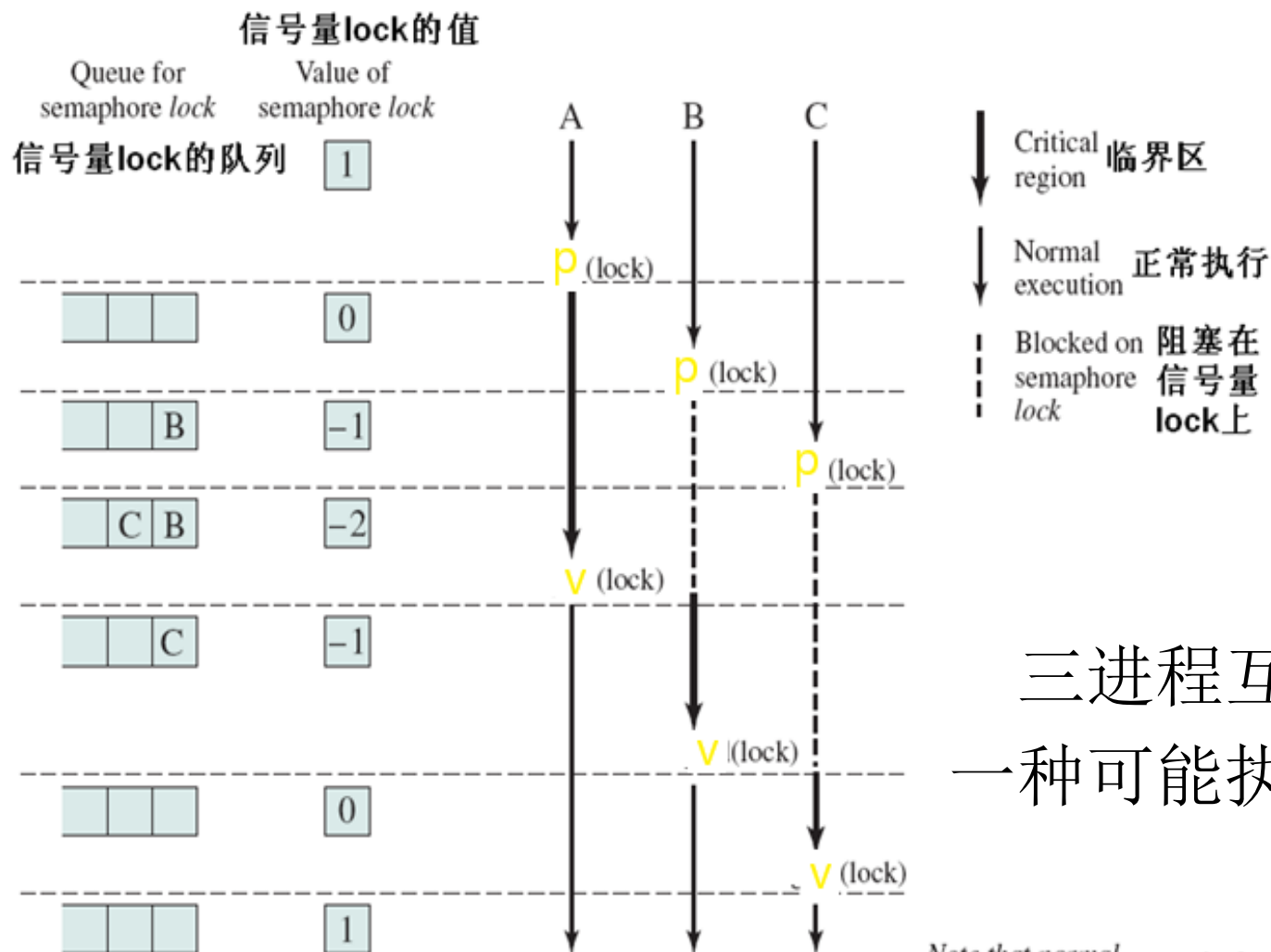
- 实现互斥
- 进程同步
- 生产者-消费者问题
- 读者-写者问题
- 其它



5.3.1 用信号量实现互斥

- n 个进程访问同一个共享资源（临界资源）
- 设置信号量 s ，初始化为1
- 每个进程进入临界区之前执行P操作
- 进程离开临界区时执行V操作
- 这样可以保证最多只有一个进程在临界区，从而实现了共享资源的互斥访问

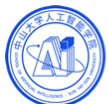
5.3.1 用信号量实现互斥（续）



三进程互斥的
一种可能执行情况

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

注意：正常执行可并行，但临界区是串行的

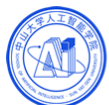


中

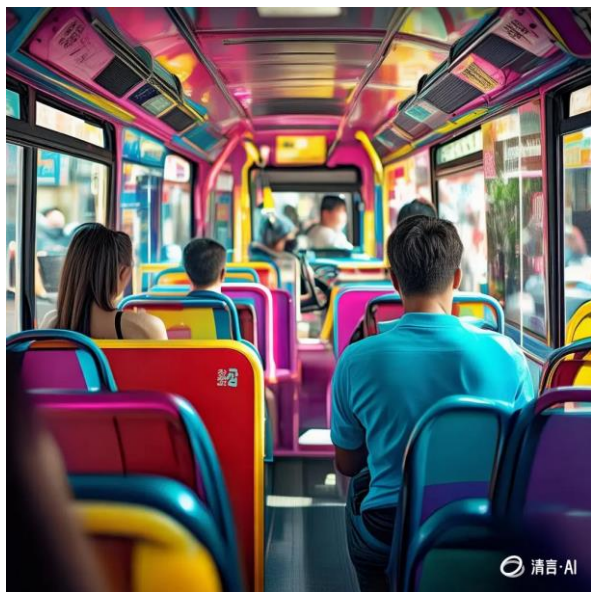
用信号量实现进程同步

■ 进程的同步（synchronization）

- 指系统中一些进程需要相互合作，共同完成一项任务：一个进程运行到某一点时要求另一伙伴进程为它提供消息，在未获得消息之前，该进程处于阻塞状态，获得消息后被唤醒进入就绪态。
- 指系统中一些进程需要相互合作，操作的执行存在时序某种的制约。



用信号量实现进程同步



司机(P1)
REPEAT

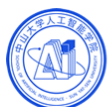
启动
正常行驶
到站停车

UNTIL FALSE

售票员 (P2)
REPEAT

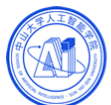
关门
售票
开门

UNTIL FALSE



用信号量解决同步关系的方案

- 两个进程P1、P2，分别有操作a和b，存在a先于b操作
- 设置信号量s，初始化0或具体资源限制值
- 进程P1的操作a后执行v操作
- 进程P2的操作b前执行P操作
- 每执行一次操作a后，才可以执行b操作，从而实现进程同步

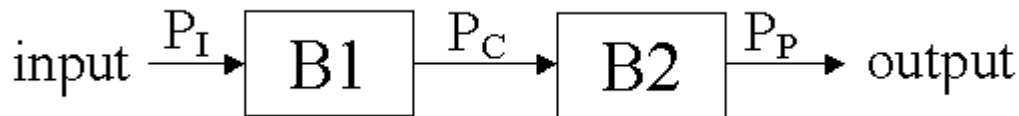


用信号量实现进程同步--举例

- 例: 有三个进程并发运行, 合作完成输入数据、计算和打印输出工作

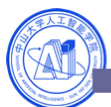
- 进程 P_I 将输入的数据写入缓冲区B1,

- 进程 P_C 读出B1中的数据, 完成计算, 把结果写入缓冲区B2



- 进程 P_P 读出B2中的结果, 打印输出

- 同步要求: (读出数据后缓冲区为空)



用信号量实现进程同步（续）

- 四个信号量：empty1、full1、empty2、full2
- 初始分别为：1、0、1、0（两个缓冲区都为空）

```
PI:  
while ( 1 ) {  
    P(empty1);  
    输入数据写到B1;  
    V(full1);  
}
```

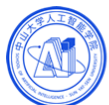
```
PC:  
while ( 1 ) {  
    P(full1);  
    从B1中读取数据;  
    V(empty1);  
    计算;  
    P(empty2);  
    结果写到B2;  
    V(full2);  
}
```

```
PP:  
while ( 1 ) {  
    P(full2);  
    读取B2中的结果  
    并输出到打印机;  
    V(empty2);  
}
```

5.3.2 生产者/消费者问题

■ 问题描述

- 若干进程通过有限的共享缓冲区交换数据
 - 一组“生产者”进程不断写入
 - 另一组“消费者”进程不断读出
 - 共享缓冲区无限/共有N个
 - 任何时刻只能有一个进程可对共享缓冲区进行操作

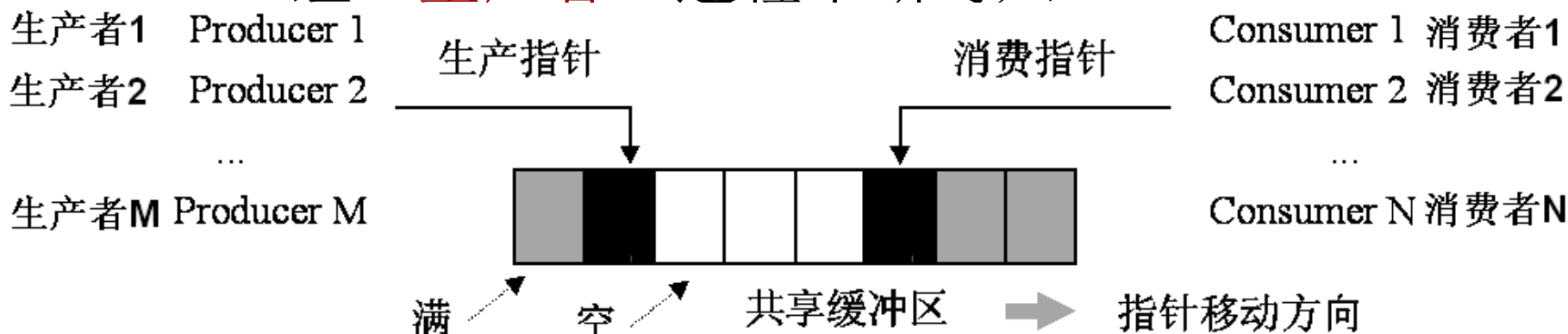


5.3.2 生产者/消费者问题

■ 问题描述

- 若干进程通过**有限的共享缓冲区**交换数据

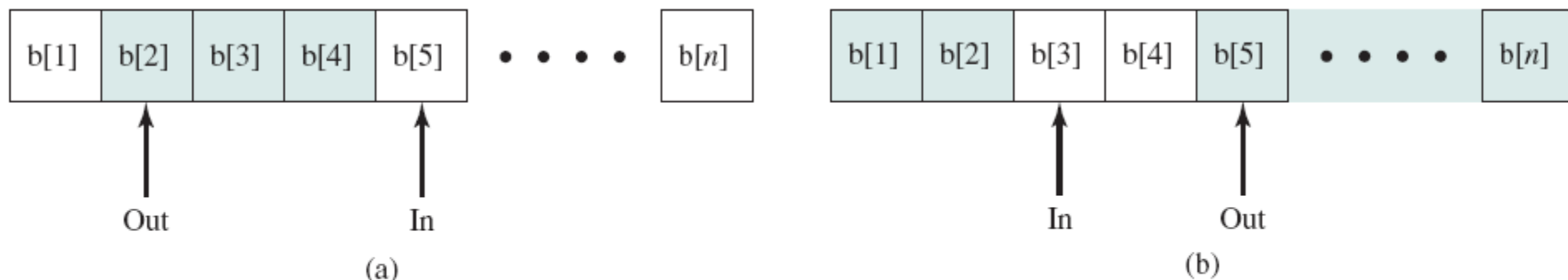
- 一组“**生产者**”进程不断写入



- **任何时刻只能有一个进程可对共享缓冲区进行操作**

有限缓冲区的生产者/消费者问题

| 对象 | 被阻塞事件 | 解除阻塞事件 |
|-----|---------|---------|
| 生产者 | 插入满缓冲区 | 消费者移出一项 |
| 消费者 | 从空缓冲区移出 | 生产者插入一项 |



Finite Circular Buffer for the Producer/Consumer Problem
生产者/消费者问题的有限循环缓冲区

有限循环缓冲区的解决方案

```
const int sizebuffer=N
semaphore n=0; /*产品数*/
semaphore s=1; /*互斥*/
semaphore e=N; /*空闲数*/
void producer() {
    while (true) {
        produce();
        p(e);
        p(s);
        append();
        v(s);
        v(n);
    }
```

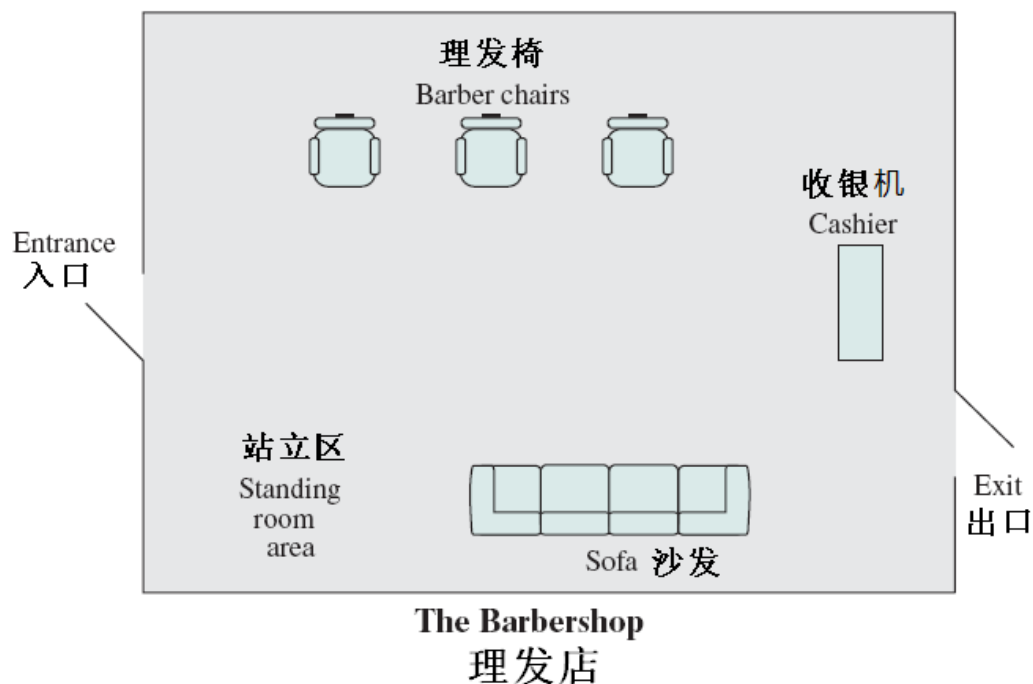
```
void consumer() {
    while (true) {
        p(n);
        p(s);
        take();
        v(s);
        v(e);
        consume();
    }
}
void main() {
    parbegin(producer, consumer);
}
```



理发店问题

■ 问题描述：

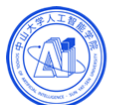
3个理发师、 3张理发椅、 一张沙发4个位、 一个收银机、 室内最多容纳20个顾客、 共有50个顾客， 有位则坐， 无位则站



5.4 管程 (monitor)

■ 动机

- 同步机制与同步策略的分离是灵活的，同时也是危险的
- 集中管理（封装）以策安全



5.4 管程

- 管程 (monitor) 是一种封装同步机制与同步策略的**程序设计语言结构**
 - Ada 95、并发Pascal、Modula-3、Java、C#、Delphi、Python、Ruby、Mesa等
- 1972年由英国计算机科学家C.A.R. Hoare和美籍丹麦计算机科学家P.B. Hansen发明

5.4.1 使用信号的管程

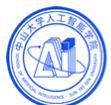
- 1974年Hoare提出的管程方案
- 1975年Hansen在并发Pascal上实现
- 主要特点：
 - 本地变量只能由管程过程访问（封装）
 - 进程通过调用管程过程进入管程（调用）
 - 每次只能一个进程执行相关管程的过程（互斥）



5.4.1 使用信号的管程

■ 主要缺陷

- 可能增加了两次多余的进程切换
- 对进程调度有特殊要求（不允许插队）



管程的结构和应用

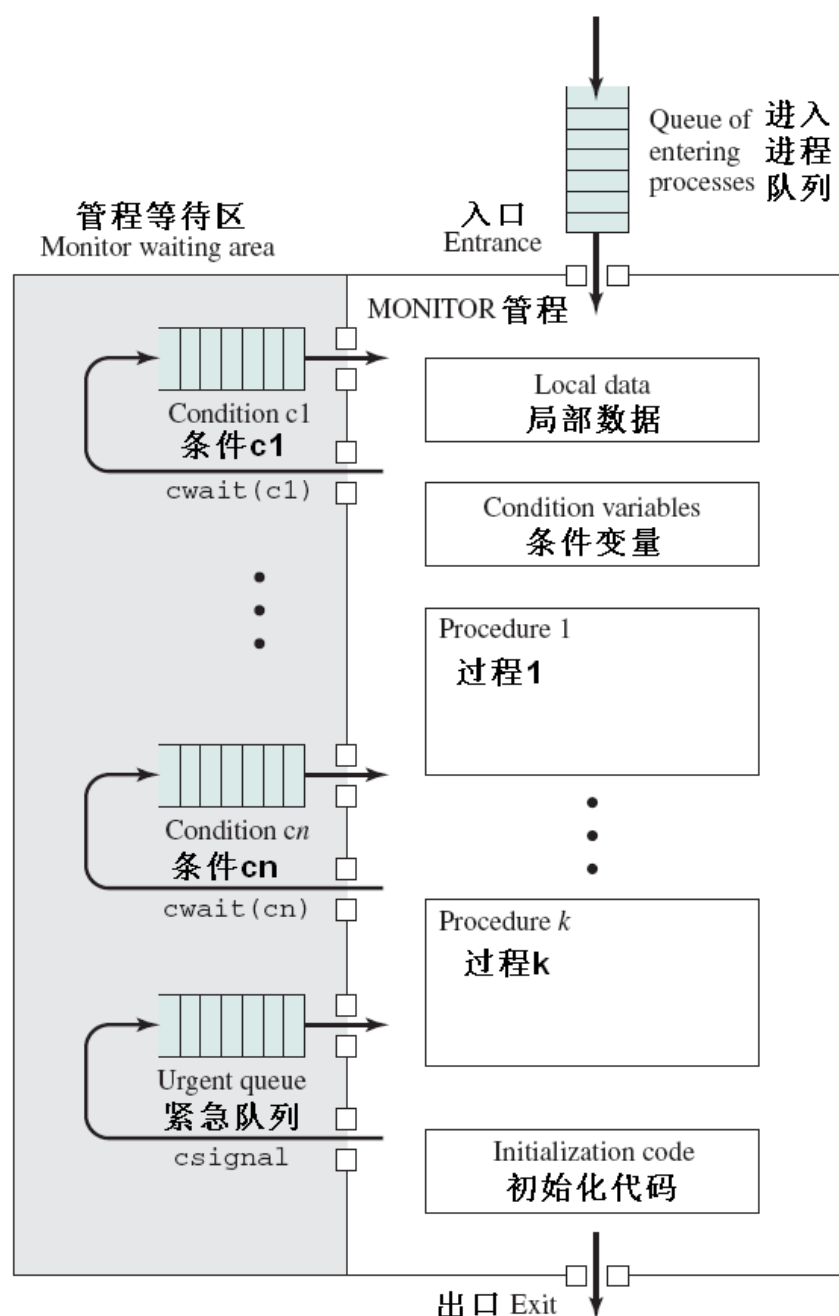
■ 管程软件模块的组成

- 若干过程
- 一个初始化序列
- 局部数据
- 条件变量

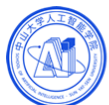
管程的条件变量定义在高

级语言中，信号量定义在

内核中



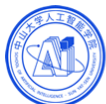
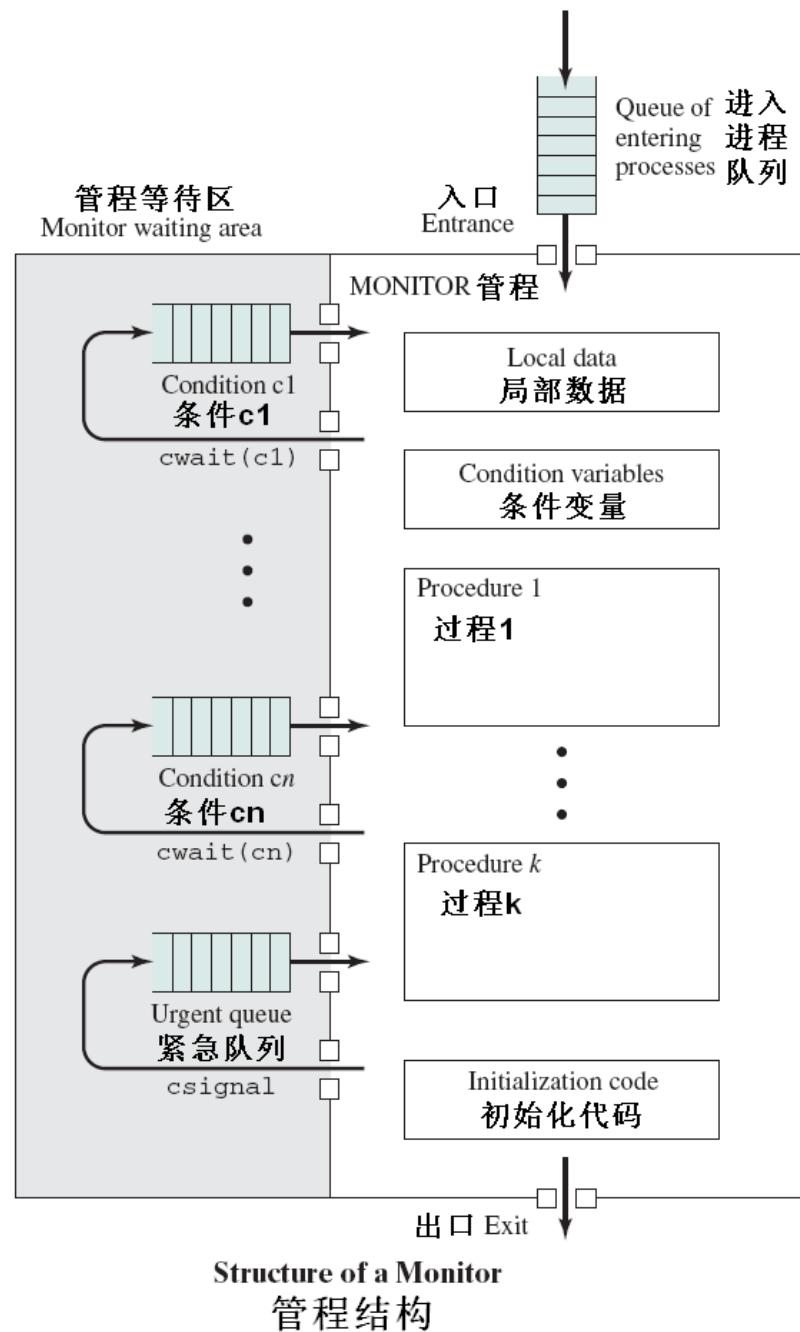
Structure of a Monitor
管程结构



管程的结构和应用

■ 管程提供的互斥机制

- 管程中的数据每次只能被一个进程访问
- 可将共享数据结构放入管程以得到保护
- 可用这些数据代表临界资源



管程的结构和应用

■ 管程对同步的支持

- 通过 `cwait(c)`、`csignal(c)` 操作管程中的条件变量实现同步

