为什么说操作系统是由中断驱动的?

第一章

operating system：操作系统

application program：应用程序

graphic user interface（GUI）：图形用户界面

simultaneous peripheral operations on line , SPOOL：假脱机

job：作业

concurrency：并发性

sharing：资源的共享性

asynchronous：异步性

virtual：虚拟性

batch processing operating system：多道批处理操作系统

time-sharing operating system：分时操作系统

real-time operating system：实时操作系统

general purpose operating system：通用操作系统

single-user　operating system：单用户操作系统

network operating system , NOS：网络操作系统

distributed operating system ,DOS： 分布式操作系统

closely coupled：紧耦合

loosely coupled：松散耦合

操作系统的作用：

① 管理系统中的各种资源

② 为用户提供友好的界面

操作系统的产生：

- 手动操作阶段
- 批处理阶段

- 批处理阶段
  - 联机批处理
  - 脱机批处理
- 执行系统阶段
  - 硬件在两个方面取得了重要的进展。一是引入通道，二是出现通道中断主机功能
  - 假脱机

通道又称I/O处理器

操作系统的完善（发展过程）：

- 多道批处理系统
- 分时系统
- 实时处理系统
- 通用操作系统

操作系统的分类：

- 多道批处理操作系统
  - 以脱机操作为标志
  - 两个标志
    - 多道
    - 成批
- 分时操作系统
  - 是以联机操作为标志，适合于程序的动态调试和修改
  - 3个重要的特性
    - 多路性
    - 交互性
    - 独占性
- 实时操作系统
  - 所谓实时，是指系统能够对外部清求做出及时的响应。
  - 2个特性
    - 及时性
    - 可靠性
  - 按其应用范围可以分为
    - 实时控制
    - 实时信息处理
- 通用操作系统
  - 同时具有分时、实时和批处理功能的操作系统
  - 可能同时存在3类任务。这3类任务通常按照其急迫程度加以分级：实时任务级别最高，分时任务级别次之，批处理任务级别最低

        - 实时任务
        - 分时任务
        - 批处理任务
- 单用户操作系统
    - 最主要的特点是单用户，即系统在同一段时间内仅为一个用户提供服务
- 网络操作系统
    - 用于实现网络通信和网络资源管理的操作系统
    - 两个目的：
        - 相互通信
        - 共享资源
- 分布式操作系统
- 多处理器操作系统
- 集群操作系统
- 云计算操作系统嵌入式操作系统
- 多媒体操作系统
- 智能卡操作系统

第五章

deadlock：死锁。一组进程中的每个进程均等待此组进程中其他进程所占有的、因而永远无法得到的资源，这种现象称为进程死锁，简称死锁。

starvation：饥饿

aging：老化

deadlock prevention：死锁预防（静态）

deadlock avoidance：死锁避免（动态）

infinite postpone：无限延迟。即使系统没有发生死锁，某些进程也可能会长时间地等待。

starve to death：饿死。等待时间给进程的推进和响应带来明显的影响时，就称发生进程饥饿。当饥饿到一定程度的进程所赋予的任务即使完成也不再具有实际意义时，称该进程被饿死。

没有时间上界的等待

- 排队等待
- 忙式等待

死锁的类型

- 竞争资源引起的死锁

- 进程间通信引起的死锁
- 其他原因引起的死锁

死锁的条件（Coffman条件）

1. 资源独占（Mutual Exclusion）

- **定义**： 一个资源在任何时刻只能由一个进程使用。这意味着，如果资源正在被一个进程使用，其他进程必须等待直到资源被释放。
- **例子**： 打印机或磁盘设备等硬件资源，通常不允许多个进程同时访问。

2. 不可剥夺（No Preemption）

- **定义**： 一旦资源被分配给一个进程，它就不能被其他进程强行夺走，只能在该进程使用完毕后主动释放。
- **例子**： 一个进程正在处理的数据不能被操作系统中断并分配给其他进程。

3. 保持申请（Hold and Wait）

- **定义**： 一个进程在请求新的资源时，可以持有它已经分配到的资源，并且不会释放这些资源，直到它再次请求资源。
- **例子**： 如果一个进程请求一个资源，但该资源当前不可用，它将继续持有它已经拥有的资源，同时等待新资源的分配。

4. 循环等待（Circular Wait）

- **定义**： 存在一组进程，其中每个进程都等待另一个进程所占有的资源，形成一个循环链。
- **例子**： 如 p1 等待 p2 持有的资源，p2 等待 p3 持有的资源，p3 又等待 p1 持有的资源，形成一个等待循环。

死锁预防

- **预先分配策略**
  - 进程在运行前一次性地向系统申请它所需要的全部资源。如果系统当前不能满足进程的全部资源请求，则不分配资源，此进程暂不投入运行。如果系统当前能够满足进程的全部资源请求，则一次性地将所申请的资源全部分配给申请进程。
  - **破坏了“保持申请”的条件**
  - 缺点
    - 资源利用率低
    - 进程在运行前可能并不知道它所需要的全部资源
- **有序分配策略**
  - 事先将所有资源类完全排序，即对每一个资源类赋予唯一的整数。规定进程必须按照资源编号由小到大的次序申请资源。
  - 破坏了“循环等待”条件
  - **优点**：相比预先分配法，资源利用率更高。
  - **缺点**：
    - 设备增加不便：新增资源类型可能需要重新编号。

- 资源浪费：进程实际使用资源的顺序可能与编号顺序不一致。
- 用户编程麻烦：必须按照规定的顺序申请资源。

安全状态

说系统处于安全状态，如果存在一个由系统中的所有进程构成的安全进程序列<p1,p2,...pn>；说一个进程序列<p1,p2,...pn>是安全的，如果对于每一个进程pi，它以后需要的资源数量不超过系统当前剩余资源数量与所有进程Pj (j < i）当前占有资源数量之和。

- 安全、不安全、死锁之间的关系：
- 死锁意味着不存在一条路线使所有进程剩余的活动能够执行完
- **不安全状态不一定是死锁状态**
- 安全序列可能有多个，进程在实际进行时并不一定沿着检测到的顺序路线进行

不安全状态但不死锁的例子：

例 5-5 考虑资源集合 $R=\{A(1), B(1)\}$，进程集合 $P=\{p_1, p_2\}$，已知进程 p_1 和进程 p_2 的活动序列 p_1：a，b，$\bar{a}$，$\bar{b}$（说明：小写字母表示申请该资源，小写字母上加一横线表示释放该资源。如 a 表示申请资源 A，$\bar{a}$ 表示释放资源 A）：p_2：b，$\bar{b}$，b，a，$\bar{b}$，$\bar{a}$。显然 p_1 和 p_2 的资源最大需求量均为 $A(1)$、$B(1)$。假定某时刻系统状态如下：

	Claim		Allocation		Need		Available	
	A	B	A	B	A	B	A	B
p_1:	1	1	1	0	0	1	0	1
p_2:	1	1	0	0	1	1		

即此时 p_1 的请求 a 被系统接受。其后系统接收到的命令有两种可能，一是 p_1 的请求 b，二是 p_2 的请求 b。假定为 p_2 的请求 b，因为 Request[2]=(0，1) ≤Need[2]=(1，1)，故该请求是合法的。又 Request[2]=(0，1) ≤Available=(0，1)，故系统当前能够满足该请求。实施分配后系统状态变化如下：

	Claim		Allocation		Need		Available	
	A	B	A	B	A	B	A	B
p_1:	1	1	1	0	0	1	0	0
p_2:	1	1	0	1	1	0		

运行安全性检测算法可以发现此时系统处于不安全状态，因而取消分配，进程 p_2 等待。实际上，如果真正实施资源分配，系统并不会进入死锁状态，因为分配资源后按照 $p_2(\bar{b})$，$p_1(b)$，$p_1(\bar{a})$，$p_1(\bar{b})$，p_2 (b)，$p_2(a)$，$p_2(\bar{b})$，$p_2(\bar{a})$的次序，两个进程可以执行完毕。这是一个 p_1 和 p_2 交叉执行的次序，而不是一个顺序执行的次序，银行家算法不能判断。这个例子验证了前面给出的论断：死锁状态是不安全状态的真子集。

银行家算法存在的问题：

死锁预防策略相比，死锁避免策略提高了资源的利用率，但是增加了系统的开销。

银行家算法是避免死锁的一个漂亮方法，但在应用时有以下几个问题。

①进程运行前需要申明所需资源的最大量，这对进程是一个负担，而且有时是困难的。

② n个进程要事先已知，实际应用中进程是动态创建动态撤销的。

③运行银行家算法进行安全性检测时，时间开销为$O(mn^2)$，由于对每个可满足的资源请求都需进行安全性检测，因而所花费的时间代价还是比较大的。

死锁和饿死的区别和联系：

联系：二者都是由于竞争资源而引起的

区别：

①从进程状态考虑，死锁进程都处于等待态。忙式等待（处于运行态或者就绪态）的进程并非处于等待态，但是却有可能被饿死。

②死锁进程等待永远不会被释放的资源，饿死进程等待会被释放但却不会分配给自己的资源，其等待时限没有上界（排队等待或忙式等待）。

③死锁一定是发生了循环等待，而饿死则不然。这也表明通过资源分配图可以检测死锁存在与否，但却不能检测是否有进程饿死。

④死锁一定涉及多个进程，而饥饿或被饿死的进程可能只有一个。

解决饥饿问题的方法

饥饿与死锁和活锁不同，它是一个可解决的问题。**饥饿主要由于资源调度策略的不公平性引起，这包括处理器资源和其他资源的分配。**

解决策略：

1. **改进资源分配算法：** 从优化资源分配算法开始，以解决不公平性问题。
2. **先到先服务算法：** 作为一种简单方法，先到先服务算法可以防止饥饿发生，但这种方法效率不高。

设计考虑： 在设计调度算法时，需要同时考虑公平性和系统效率。

推荐方法：

- 引入"老化"作为调度算法的一个参数。**老化指的是进程等待某种资源的时间。**
- 调度算法应考虑多个参数，特别是老化，以有效解决饥饿问题。

具体实施：

- 进程等待CPU资源时，随着等待时间的增加，其优先级应动态提升。
- 优先级提升的进程会逐步超过当前运行进程的优先级，从而获得运行机会。
- 这种方法可以保证不会发生饥饿和饿死情况。

第八章

file：文件。

file system：文件系统。**文件与管理信息资源的程序集合**称为文件系统。（从系统层次的角度来看，文件系统位于设备管理的上层）

sequeniial access：顺序访问。按照从前到后的次序依次存取文件的各个信息项。

key：键。是文件信息项中的一个域。按键随机访问就是按照信息项中的某个键值随机地存取文件的记录。

file control block，FCB：文件控制块，是标志文件存在的数据结构，其中包含系统对文件进行管理所需要的全部信息。

对于文件的访问终将落实到对于磁盘等设备的输入输出操作上，因而从软件层面上来说，文件系统位于设备管理之上。

UNIX系统中，将文件分为3类：

1. **普通文件（Regular Files）**：
   - 用途：保存数据、程序代码、音乐、图像等。
   - 特点：是UNIX系统中最常见的文件类型，用于存储信息。
2. **目录文件（Directory Files）**：
   - 用途：保存文件系统内的文件和目录的描述信息。
   - 特点：目录文件允许用户通过层级结构来组织和访问文件系统。
3. **特殊文件（Special Files）**：
   - 用途：对应系统中的各种设备，如打印机、硬盘、终端等。
   - 特点：UNIX系统将设备作为文件处理，使得设备的操作和管理更加统一和方便。

将设备作为文件处理是UNIx系统的一个标志性成功特点

目录项：

每一个文件都有一个文件控制块，它们被保存于外存空间中。当欲访问一个文件时，应当能够根据文件名称找到它所对应的文件控制块。那么，文件控制块是如何保存于外存中的呢？它是作为目录项存储于目录文件中的。因而，文件控制块也被称为目录项。

第九章

Polling：探询。探询（Polling）又称程序控制输入输出（progra？ed 1 / 0），是最早的输入输出控制方式，处理器代表进程向相应的设备模块发出输入输出请求，然后处理器反复查询设备状态，直至输入输出完成。其缺点是忙式等待，处理器与设备完全串行工作，由于设备速度远远低于处理器速度，忙式等待过程将消耗大量处理器时间。

寻道时间（seek time),这是指将磁盘引臂移动到指定柱面所需要的时间；

旋转延迟(rotational delay),这是指定扇区旋转到磁头下的时间；

传输时间(transfer time),这是指读写一个扇区的时间。

track discrimination：磁道歧视。假设某一时刻的外磁道请求不断，则内磁道请求可能长时间得不到满足，这种现象称为“磁道歧视”

前述扫描和循环扫描算法基本上是公平的，而且效率较高。不过，若在一段时间内同一柱面的访问请求不断，则磁头会停在一个磁道上不动，称为“磁头黏性”(magnetic head stickiness)

选择题

在进程运行时发生的如下事件中：
①时钟中断
②调用访管输入
③执行非法指令
④I/O中断
一定进行进程切换的事件是（②③）

首先要明晰任务调度的时机：**当系统发生硬件中断、系统调用或者时钟中断时，就有可能发生进程的切换。**

而其中中断的分类为

- （1）外中断（又称为中断或异步中断）
  ①泛指来自处理器之外的中断信号，包括外部设备中断、时钟中断、键盘中断和它机中断等；
  ②分为可屏蔽中断和不可屏蔽中断；
  ③高优先级中断可以部分或全部屏蔽低级中断。
- （2）内中断（又称为异常或同步中断）
  ①泛指来自处理器内部的中断信号，往往由于在程序执行过程中，发现与当前指令关联的、不正常的或错误的事件；
  ②细分为访管中断、硬件故障中断、程序性异常；
  ③内中断不能屏蔽，一旦出现应立即响应并处理。

所以上述中一定发生进程切换的要选择内中断也就是②③

下列选项中，不可能在用户态发生的事件是（C）。
A 系统调用
B 外部中断
C 进程切换
D 缺页

1.系统调用可能在用户态和内核态发生，系统调用把应用程序的请求（用户态的请求）传入内核，由内核（内核态）处理请求并将结果返回给应用程序（用户态） 用户态->核心态
2.中断的发生与CPU当前的状态无关，既可以发生在用户态，又可以发生在内核态，因为无论系统处于何种状态都需要处理外部设备发来的中断请求。
3.进程切换在核心态下完成，不能发生在用户态。原因：需要调度处理器和系统资源，为保证系统安全
4.缺页（异常）也是用户态->内核态
ABD（系统调用中断异常）都是用户态转向内核态，而进程切换只能发生在内核态

死锁

死锁避免

资源分配

算法 5-1 银行家算法——资源分配算法。

① 如果 Request[i]≤Need[i]，则转步骤②；否则进程申请资源量超过所声明的最大资源需求量，带错返回。

② 如果 Request[i]≤Available，则转步骤③；否则当前无法满足本次申请，进程 p_i 必须等待。

③ 假设系统分配资源，将相应的数据结构改为：

Available = Available – Request[i];

Allocation = Allocation + Request[i];

Need[i] = Need[i] – Request[i];

④ 如果上述分配所导致的新状态是安全的，则转步骤⑤；否则取消分配。

Available = Available + Request[i];

Allocation = Allocation – Request[i];

Need[i] = Need[i] + Request[i];

进程 p_i 等待。

⑤ 确认分配，进程 p_i 继续执行。

为了进行安全性检查，需要定义以下数据结构。

int Work[m]：工作变量，记录可用资源。

int Finish[n]：工作变量，记录进程是否可以执行完。

安全性检测算法

算法 5-2 银行家算法——安全性检测算法。

① Work = Available；Finish = false。

② 寻找满足以下条件的 i：

Finish[i]==false；Need[i]≤Work[i]；

如果不存在，则转步骤④。

③ Work = Work + Allocation[i]；Finish[i] = true；

转步骤②。

④ 如果对于所有 i，Finish[i] = true，则系统处于安全状态，否则系统处于不安全状态。

银行家算法的时间复杂度主要是安全性检测算法的复杂度，步骤② 、 ③最多执行n次，**每次最多扫描数组Need的n个元素，由于资源类别数为m**，故银行家算法的时间复杂度为$O(mn^2)$,借助图论可将其复杂度降至$O(n^2)$。

死锁检测

① Available：长度为 m 的向量，记录当前各个资源类中空闲资源实例的个数。

② Allocation：$m \times n$ 的矩阵，记录当前每个进程占有各个资源类中资源实例的个数。

③ Request：$m \times n$ 的矩阵，记录当前每个进程申请各个资源类中资源实例的个数。如果 Request[i，j]=k，则进程 p_i 申请资源类 r_j 中的 k 个资源实例。

两个向量间的关系和赋值操作的定义如前所示，为了表达简洁，将矩阵 Allocation 和 Request 的行看作向量，并且分别表示为 Allocation[i]和 Request[i]。

算法 5-3 死锁检测算法。

① 令 Work 和 Finish 分别是长度为 m 和 n 的向量，初始时设置：

（a）Work = Available。

（b）对于所有 i=1，2，…，n，如果 Allocation[i]≠0，则 Finish[i] = false，否则 Finish[i] = true。

② 寻找满足下述条件的下标 i：

（a）Finish[i] = false。

（b）Request[i]≤Work。

如果不存在满足上述条件的 i，则转步骤④。

③ Work = Work+Allocation[i]；

Finish[i] = true；

转步骤②。

④ 如果存在 i，1≤i≤n，Finish[i]=false，则系统处于死锁状态，且进程 p_i 参与了死锁。

例题：

例 5-6 设系统中有 3 个资源类{A，B，C}。资源类 A 中有 11 个实例，资源类 B 中有 6 个实例，资源类 C 中有 7 个实例。又设系统中有 5 个进程{p_0，p_1，…，p_4}。假定某时刻系统中资源分配与申请情况如下：

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
p_0	0	1	0	0	0	0	0	2	1
p_1	3	0	0	1	1	2			
p_2	4	1	4	0	0	0			
p_3	2	2	0	1	0	0			
p_4	1	0	2	0	1	3			

此时，系统不处于死锁状态，因为运行上述死锁检测算法可以得到一个进程序列<p_0，p_2，p_3，p_1，p_4>，它将使 Finish[i]=true，对于所有 1≤i≤n 成立。

假定现在进程 p_2 发出请求(0，0，2)，即申请资源类 C 中的两个资源实例。请求矩阵 Request 变化如下：

	Request		
	A	B	C
p_0	0	0	0
p_1	1	1	2
p_2	0	0	2
p_3	1	0	0
p_4	0	1	3

此时，系统处于死锁状态，参与死锁的进程构成的集合为{p_1，p_2，p_3，p_4}。

I/O

磁盘访问

例 9-1 设有一个只有单一移动磁头的磁盘，磁道由外向内编号为 0，1，2，…，199，磁头移动一个磁道所需时间为 1 ms，每个磁道有 100 个扇区，磁盘转速 6 000 rpm。采用循环扫描算法，当前引臂位置处于第 100 磁道，当前移动方向为由外向内，并规定引臂向内扫描时为路径请求服务。对于磁道请求 120、85、70、30，每个请求访问对应磁道上的一个扇区，试问：① 给出引臂移动序列，计算引臂移动量和寻道时间，忽略启动时间；② 计算平均旋转延迟时间；③ 计算传输时间；④ 计算所有访问处理时间。

解答：

① 磁盘引臂移动序列为 100→120→30→70→85，跨越磁道数 20+90+40+15=165。共需寻道时间 165×1 ms=165 ms。

② 1 次访问磁盘的旋转延迟为 $T_r=1/(2r)=1/(2\times(6\ 000\ \text{rpm}))=1/(2\times(100\ \text{rps}))=5$ ms，4 次访问磁盘的旋转延迟为 4×5 ms=20 ms。

③ 1 次访问磁盘的传输时间为 $T_t=1/(rM)=1/((6\ 000\ \text{rpm})\times 100)=1/((100\ \text{rps})\times 100)=0.1$ ms，4 次访问磁盘的传输时间为 4×0.1 ms=0.4 ms。

④ 所有访问处理时间为 165 ms+20 ms+0.4 ms=185.4 ms。

在UNIX系统中，进程P部分程序如下：

```
……
int pid1,pid2;
int fd[2];
char buf[50];

if(pid1=fork()==0)
{
    close(fd[1]);
    read(fd[0],buf,6);
    sleep(100);
    exit(1);
}

if(pid2=fork()==0)
{
    close(fd[0]);
    write(fd[1],"Hello");
    sleep(100);
    exit(2);
}

close(fd[0]);
close(fd[1]);
……
```

画图说明上述程序在exit执行前，系统中u_file表、file表、inode表的主要内容及表之间的联系情况，以及buf的内容

互斥算法

软件实现

尝试1：使用全局变量`turn`实现互斥【单标志法】

- **代码**：

```
int turn;   //turn表示当前允许进入临界区的进程号
P0: do{
      while (turn==1) ;
      临界区代码
      turn=1;
      其余代码
}while(1);
P1: do{
      while(turn==0);
      临界区代码
      turn=0;
      其余代码
}while(1);
```

- **问题**：**不满足进展性原则。**如果P0和P1同时到达它们的while循环，它们将交替进入临界区。该算法可确保每次只允许一个进程进入临界区。但两个进程必须交替进入临界区，**若某个进程不再进入临界区，则另一个进程也将无法进入临界区(违背“空闲让进”)。**这样很容易造成资源利用不充分。**比如若P0顺利进入临界区并从临界区离开，则此时临界区是空闲的，但P1并没有进入临界区的打算，turn=1一直成立，P0就无法再次进入临界区(一直被 while死循环困住)。**

尝试2：【双标志先检查法】

- **代码**：

```
Boolean flag[2];
P0: do{
        while (flag[1]); //①
        flag[0]=true;   //③
        临界区
        flag[0]=false;
        其余代码
}while(1);
P1: do{
        while (flag[0]); //②
        flag[1]=true;   //④
        临界区
        flag[1]=false;
        其余代码
}while(1);
```

- **思想**：在每个进程访问临界区资源之前，先查看临界资源是否正被访问，若正被访问，该进程需等待；否则，进程才进入自己的临界区。
- **优点**：不用交替进入，可连续使用
- **问题**：不满足互斥性原则。如果按照序列①②③④执行，`flag[0]`和`flag[1]`同时为`false`，两个进程可以同时进入临界区，违反了互斥性原则。【进入区的检查和上锁并不是"一气呵成"的，先检查后上锁可能发生进程切换】

尝试3：【双标志后检查法】

- **代码**：

```
boolean flag[2]; (false,false)
do{
        flag[0]=true;
        while (flag[1]);
        临界区代码
        flag[0]=false;
        其余代码
}while(1);
do{
        flag[1]=true;
        while(flag[0]);
        临界区
        flag[1]=false;
        其余代码
}while(1);
```

- **思想**：先将自己的标志设置为 TRUE,再检测对方的状态标志，若对方标志为TRUE，则进程等待；否则进入临界区。【先上锁后检查】
- **问题**：不满足进展性原则（违背"空闲让进"）。如果`flag[0]`和`flag[1]`同时为`true`，两个进程都会在while循环中无限期地等待，导致饥饿现象。

当while条件成立时，进程P不能向前推进，而在原地踏步，这种原地踏步被称为"忙式等待"。**不进入等待状态的等待**称为**忙式等待**（busy waiting）。注意，**这里虽然用了"等待"一词，但是进程并未真正进入等待状态，实际状态为"运行"或者"就绪"**。忙式等待空耗处理器资源，因而是低效的。

针对两个进程的互斥算法

1、 Dekker算法

```
int flag[2]; (init 0)   //初值为0
int turn; (0 or 1)  //初值为0或1

P0:
do{
    flag[0]=1;
    while(flag[1])
        if(turn==1){
            flag[0]=0;
```

```
            while (turn==1)
                skip;
            flag[0]=1;
        }
    临界区
    turn=1;
    flag[0]=0;
    其余代码
}while(1);

P1:
do{
    flag[1]=1;
    while(flag[0])
        if(turn==0){
            flag[1]=0;
            while (turn==0)
                skip;
            flag[1]=1;
        }
    临界区
    turn=0;
    flag[1]=0;
    其余代码
}while(1);
```

①考察互斥性。只需证明当一个进程进入其临界区后，另一进程不能进入其临界区。不失一般性，假定P0已经进入其临界区，此时flag[0]为1成立，P1欲进入临界区必将在其外层while循环处忙式等待，因而满足互斥性。.

②考察进展性。若只有一个进程（设为P0）欲进入其临界区，由于flag [1]为0，P0结束外层while循环，进入其临界区。若两个进程都想进入临界区，假设tum的值为0，进程P1的if条件成立，将自己的flag[1]置为0，并动态等待P0 。P0获得处理器资源运行时，检测flag[1]不成立，结束外层while循环，进入临界区，因而满足进展性。

③考察有限等待性。假设P0处于临界区中，P1正在执行其entry section代码试图进入其临界区。 P0离开临界区时，将tum的值置为1，flag[0]的值置为0，这将使P1的内层while循环条件不成立。若尸I在判断外层while循环条件之前P0没有再次提出进入临界区的请求，则flag [1]的值为0，P1结束外层while循环进入其临界区；反之，若P1判断外层while循环条件之前P0再次执行entry section代码，则会将flag[0]再次置为1，但是因为flag[1]条件和turn=1条件成立，P0将在其flag[0]标志置为0后忙式等待P1，直至P1进入并离开其临界区。因而P1在P0再度进入临界区之前，必能得到进入临界区的机会。

2、Peterson算法

```
boolean flag[2];    //表示进入临界区意愿的数组，初始值都是false
int turn;           //turn表示优先让哪个进程进入临界区（表达“谦让”）

P0:
Do{
    flag[0]=true;
    turn=1;
    while (flag[1] && turn==1);
    临界区
```

```
    flag[0]=false;
    其余代码
}while(1);

P1:
Do{
    flag[1]=true;
    turn=0;
    while (flag[0] && turn==0);
    临界区
    flag[1]=false;
    其余代码
}while(1);
```

①考察互斥性。不失一般性，假定P0已经进入其临界区，须证P1必不能进入其临界区。P0进入临界区时，其while循环条件必不成立。这有3种情况：

- （a）flag[1]不成立
- （b）turn==1不成立
- （c）二者都不成立。

对于情形(a)，说明P1尚未提出进入临界区的请求，以后该进程执行其entry section代码时，将turn赋值为0的动作必在P0将turn赋值为1的动作之后执行；
对情形(b),表明P1将turn赋值为0的动作必在P0将其赋值为1的动作之后完成；
情形(c)是不可能出现的，因为这既要求P1还没有执行entry section,又要求P1执行完turn=0语句。
上述可能的两种情况，P1执行while循环时turn==0成立。又由于P0已经在其临界区内，flag[0]=1必然成立，故P1将在其while循环处忙式等待，因而满足互斥性。

②考察进展性。若只有一个进程想进入其临界区，设P0想进入、P1不想进入，则flag[1]的值为0，P0的循环条件不成立，结束循环进入其临界区。若两个进程都想进入临界区，turn==0与turn==1必有一个不成立，因而必有一个进程结束while循环进入其临界区，因而满足进
展性。

③考察有限等待性：假定P0在其临界区内，P1提出进入临界区并在其while循环处忙式等待，当P0离开临界区时，将flag[0]的值置为0，若此时P1获得执行机会，将检测到flag[0]不成立从而进入临界区：若在P1获得处理器运行之前P0再次提出进入临界区的请求，将flag [0]的值置为1 , turn的值置为1，然后在while循环处忙式等待，而以后P1获得运行机会时检测到turn==0不成立，进入其临界区，所以P1在P0再次进入临界区之前一定能够进入临界区，因而满足有限等待性。

【但没有实现让权等待】

硬件实现

（1）中断屏蔽方法（硬件提供“关中断”和“开中断”指令）

当一个进程正在执行它的临界区代码时，防止其他进程进入其临界区的最简方法是关中断。因为 CPU 只在发生中断时引起进程切换，因此屏蔽中断能够保证当前运行的进程让临界区代码顺利地执行完，进而保证互斥的正确实现，然后执行开中断。其典型模式为

```
do{
    ……
    关中断；
    临界区；
    开中断；
    ……
}while(1);
```

这种方法限制了处理机交替执行程序的能力，因此执行的效率会明显降低。对内核来说，在它执行更新变量或列表的几条指令期间，关中断是很方便的，但将关中断的权力交给用户则很不明智，若一个进程关中断后不再开中断，则系统可能会因此终止。

- **开关中断只在单CPU系统中有效。**关中断方法不适用于多CPU系统,因为关中断只能保证CPU不由一个进程切换到另外一个进程,从而防止多个进程并发地进入公共临界区域。但即使关中断后,不同进程仍可以在不同CPU上并行执行关于同一组共享变量的临界区代码.
- **影响并发性。**中断屏蔽期间，系统无法进行进程调度，导致CPU利用率降低，影响并发性。
- 只适用于内核级：因为开关中断命令只能由内核调用，给用户会出问题

（2）硬件提供“测试并建立”指令

TestAndSet 指令：这条指令是原子操作，即执行该代码时不允许被中断。其功能是读出指定标志后把该标志设置为真。指令的功能描述如下：

```
boolean TestAndSet(int *lock){
    int old;
    old=*lock;
    *lock=1;
    return old;
}
```

可以为每个临界资源设置一个共享布尔变量lock,表示资源的两种状态；true 表示正被占用，初值为 false。进程在进入临界区之前，利用 TestAndSet 检查标志lock,若无进程在临界区，则其值为 false,可以进入，关闭临界资源，把 lock 置为true,使任何进程都不能进入临界区；若有进程在临界区，则循环检查，直到进程退出。利用该指令实现互斥的过程描述如下：

```
do{
    while TestAndSet(&lock)
        skip;
    进程的临界区代码段；
    lock=0;
    进程的其他代码；
}while(1);
```

对一组公共变量，int lock;（初始=0）； Pi进入：While test_and_set(&lock); Pi离开：lock=0;

满足互斥性，进展性，不满足有限等待性。这种方法简单，但**可能导致忙式等待**，即进程不断检查锁变量，即使在临界区未被占用时也会占用CPU资源。

执行该指令的CPU将**会锁住内存总线**（memory bus），所以在该指令执行完成之前**其他CPU是无法访问内存**的，所以**在多CPU下也可以使用**

改进

为了使其满足有限等待性，还需要引入新的变量。下面给出一个满足进程互斥全部条件的算法

算法4-6 基于“测试并设置”指令的公平性硬件互斥算法。

```
//全局变量
int waiting[n]; //初始值为0
int lock;

//局部变量
int j;
int key;

do{
    waiting[i]=1;
    key=1;
    while(waiting[i]&&key)
        key=test_and_set(&lock);
    waiting[i]=0;
    临界区
    j = (i+1)%n ;
    while((j!=i)&&(!waiting[j]))do
        j=(j+1)%n;
    if(j==i)
        lock = 0 ;
    else
        waiting[j]= 0;
    其余代码
} while(1) ;
```

P/V

哲学家

```

```

制盒生产线上有一箱子,其中有N个位置(N≥2),每个位置可放一盒体或一盒盖又设有三个工人 P1.P2,P3 其活动分别为: P1: 加工一盒体; 盒体放入箱中; P2; 加工一个盒盖: 盒盖放入箱中; P3: 箱中取一盒 体; 箱中取一盒盖; 组成一盒子: 用管程实现三个工人的合作

```
Type Box=MONITOR;// 定义一个管程Box
enum item {body, cap, null};// 定义一个枚举类型item，表示箱子中可能的物品状态
Var box:array[0..N] of item;// 定义一个数组box，用于存放箱子中的物品，每个位置可以是body、cap或null
count1, count2: integer;// 定义两个计数器，分别用于记录箱子中盒体和盒盖的数量
```

```
full, bodyempty, capempty: condition;// 定义三个条件变量，用于同步工人的等待和通知
n, out: integer;// 定义两个整型变量，用于操作箱子中物品的位置
Define putin-body, putin-cap, takeout-body, takeoutcap;// 定义四个过程：放入盒体、放入盒盖、取出盒体、取出盒盖

// 实现放入盒体的过程
Procedure putin-body(x: body) {
    // 如果箱子已满或盒体数量过多，则等待
    if (count1 + count2) >= N then wait(full);
    if count1 > N - 2 then wait(full);

    // 将盒体放入箱子的空位置
    count1 := count1 + 1;
    for i = 1 to N {
        if box[i] = null then break;
    }
    in := i;
    box[in] := x;

    // 通知可能在等待盒体的工人
    signal(bodyempty);
}

// 实现放入盒盖的过程
Procedure putin-cap(y: cap) {
    // 如果箱子已满或盒盖数量过多，则等待
    if (count1 + count2) >= N then wait(full);
    if count2 > N - 2 then wait(full);

    // 将盒盖放入箱子的空位置
    count2 := count2 + 1;
    for i = 1 to N {
        if box[i] = null then break;
    }
    in := i;
    box[in] := y;

    // 通知可能在等待盒盖的工人
    signal(capempty);
}

// 实现取出盒体的过程
Procedure takeout-body(x: body) {
    // 如果没有盒体可取，则等待
    if count1 < 1 then wait(bodyempty);

    // 从箱子中取出盒体
    count1 := count1 - 1;
    for i = 1 to N {
        if box[i] = body then break;
    }
    out := i;
    x := box[out];

    // 通知可能在等待空位的工人
    signal(full);
```

```
}

// 实现取出盒盖的过程
Procedure takeout-cap(y: cap) {
    // 如果没有盒盖可取，则等待
    if count2 < 1 then wait(capempty);

    // 从箱子中取出盒盖
    count2 := count2 - 1;
    for i = 1 to N {
        if box[i] = cap then break;
    }
    out := i;
    y := box[out];

    // 通知可能在等待空位的工人
    signal(full);
}
```

完善讲稿中独木桥问题的第二种解法：每方通过K人后，轮到对方。

目前的解法存在问题：

```
    假定当前西侧过桥者全部通过；恰巧此时东侧没有过桥者等待；如果后续西侧先来过桥者A，其无法马上过桥，必须要等到东侧出现一个人过桥后，A才能过桥。
```

下面解法基础上，进行修正，通过引入裁判程序（定时的红绿灯），避免上述问题。

要求：先语言说明算法的基本逻辑，再写具体算法，并给出适当注释。

算法基本逻辑：

- 引入一个裁判程序
  - 无限循环，模拟交通灯定时变换。
  - `T`累加时间，当时间达到`duration`时，通过异或操作`direction ^= 1`切换过桥方向，并重置`T`。
  - 根据`direction`的值，如果西侧过桥（`direction == 0`），并且有西侧过桥者等待（`w_count > 0`），则通过`V(wq)`允许西侧过桥者开始过桥，并减少`w_count`。
  - 同理，如果东侧过桥（`direction == 1`），则允许东侧过桥者开始过桥。

```
semaphore wq, eq;  // 初始值都为0，用于控制西侧和东侧的过桥者
semaphore mutex;   // 初始值为1，用于保护临界区
int w_count, e_count;  // 初始值都为0，表示西侧和东侧等待过桥的人数
int count;  // 初始值为0，表示当前某侧连续过桥者数量
int direction;  // 西为0，东为1，表示当前允许过桥的方向
int duration;  // 60，代表每60秒改变红绿灯
int T;  // 初始值为0，表示红绿灯的计时器

```

```
// 裁判程序，用于定期改变红绿灯的方向
void referee(){
    while(1){
        T = T + getElapsed();
        if(T >= duration){  // 定时切换红绿灯的方向
            direction ^= 1;
            T = 0;
        }
        P(mutex);
        if(direction == 0){  // 如果当前允许西侧过桥
            if(w_count){  // 如果西侧有等待过桥的人
                V(wq);  // 唤醒一个西侧的过桥者
                w_count--;  // 减少西侧等待过桥的人数
            }
        }
        else{  // 如果当前允许东侧过桥
            if(e_count){  // 如果东侧有等待过桥的人
                V(eq);  // 唤醒一个东侧的过桥者
                e_count--;  // 减少东侧等待过桥的人数
            }
        }
        V(mutex);
    }
}

// 西侧过桥者
void w_to_e(){
    while(1){
        P(mutex);
        w_count++;  // 增加西侧等待过桥的人数
        V(mutex);
        P(wq);  // 等待过桥的许可
        过桥
        P(mutex);
        w_count--;  // 减少西侧等待过桥的人数
        if(w_count == 0 || count == k){  // 如果西侧没有等待过桥的人，或者已经连续过桥k个人
            direction = 1;  // 切换红绿灯的方向为东侧
            count = 0;  // 重置连续过桥的人数
            while(count < k && e_count > 0){  // 如果东侧有等待过桥的人
                count++;  // 增加连续过桥的人数
                e_count--;  // 减少东侧等待过桥的人数
                V(eq);  // 唤醒一个东侧的过桥者
            }
        }
        V(mutex);
    }
}

// 东侧过桥者
void e_to_w(){
    while(1){
        P(mutex);
        e_count++;  // 增加东侧等待过桥的人数
        V(mutex);
        P(eq);  // 等待过桥的许可
```

```
        过桥
        P(mutex);
        e_count--;  // 减少东侧等待过桥的人数
        if(e_count == 0 || count == k){  // 如果东侧没有等待过桥的人，或者已经连续过桥k个人
            direction = 0;  // 切换红绿灯的方向为西侧
            count = 0;  // 重置连续过桥的人数
            while(count < k && w_count > 0){  // 如果西侧有等待过桥的人
                count++;  // 增加连续过桥的人数
                w_count--;  // 减少西侧等待过桥的人数
                V(wq);  // 唤醒一个西侧的过桥者
            }
        }
        V(mutex);
    }
}

```