

# 死锁 (deadlock) -1

教师：姜博

E-Mail: [gongbell@gmail.com](mailto:gongbell@gmail.com)

# 内容提要

- 死锁的概念
- 处理死锁的基本方法
  - 鸵鸟算法
  - 死锁检测与恢复
  - 死锁预防
  - 死锁避免
- 小结

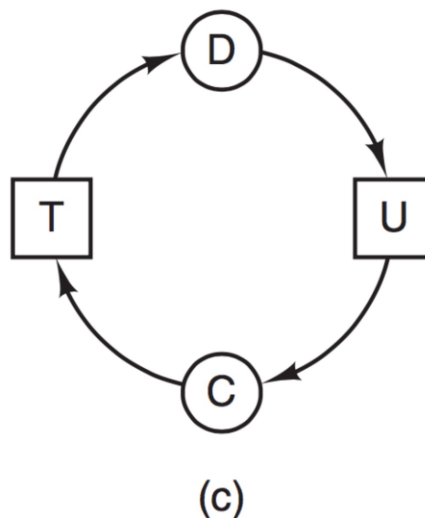
# 死锁检测

- 根据资源的请求和分配信息，利用某种算法对这些信息加以检查，以判断是否存在死锁。死锁检测算法主要是检查是否有循环等待。
- 资源分配图法
- 资源向量法

# 资源分配图/进程-资源图

## ■ 主要标记及含义

- 圆形：进程
- 方形：资源
- 从资源节点到进程节点的有向边：资源已被进程占用
- 从进程节点到资源节点的有向边：进程正在请求该资源



**Figure 6-3.** Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

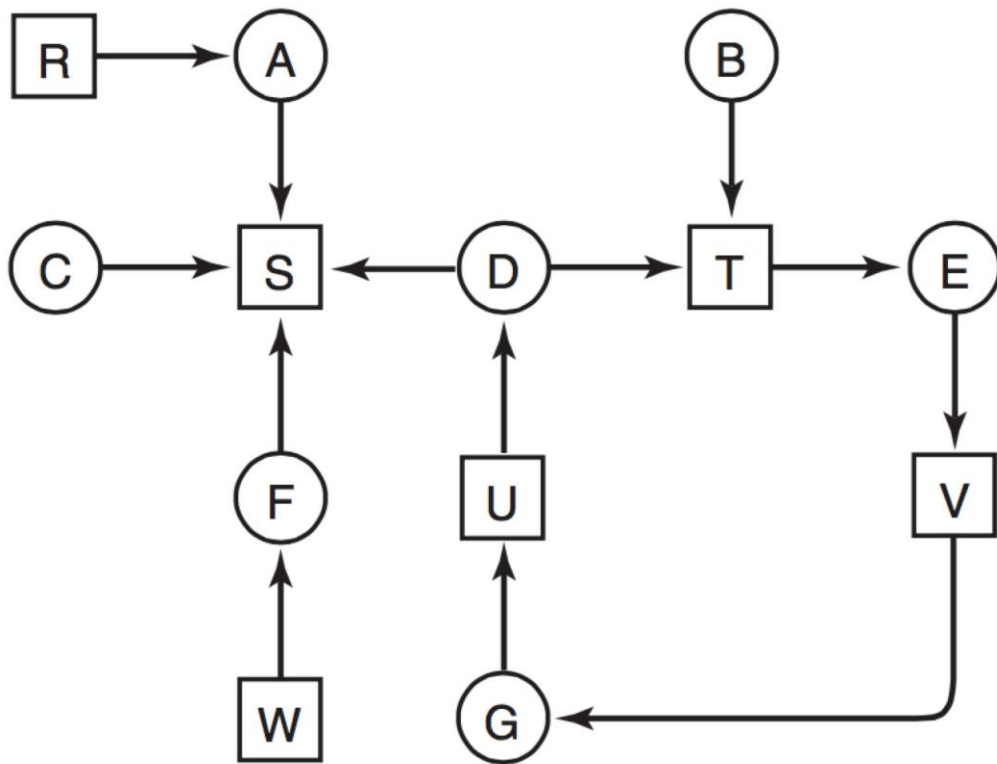
# 资源分配图(resource allocation graph)算法

- 每类资源一个的死锁检测
- 按照资源请求和释放的序列对图进行操作，并在每步操作后检查是否存在环。
- 可用作分析是否存在死锁的工具

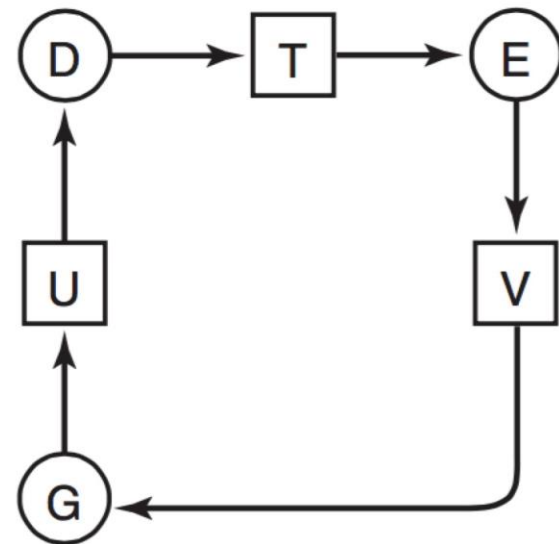
系统中是否有死锁？包含了那些进程？

- 1 进程A 已获取资源R，请求资源S
- 2 进程B未获取任何资源，请求资源T
- 3 进程C未获取任何资源，请求资源S
- 4 进程D获取了资源U，请求资源S和T
- 5 进程E获取了资源T，请求资源V
- 6 进程F 获取了资源W，请求资源S
- 7 进程G获取资源V，请求资源U

# 资源分配图(resource allocation graph)



(a)



(b)

**Figure 6-5.** (a) A resource graph. (b) A cycle extracted from (a).

# 资源向量(矩阵)算法

- 每类资源多个的死锁检测
- E: 存在资源向量 (existing resource vector): 表示各类资源存在的总量。
- A: 可用资源向量(available resource vector):表示当前未分配可使用的资源数。
- C: 当前分配矩阵(current allocation matrix): 第i个行向量对应第i个进程已经分配到的各类资源数量。
- R: 请求矩阵(request matrix): 第i个行向量表示进程i所需要的资源数量。

$$\text{恒等式: } \sum_{i=1}^n C_{ij} + A_j = E_j$$

# 资源向量(矩阵)算法


- 每类资源多个的死锁检测

Resources in existence  
( $E_1, E_2, E_3, \dots, E_m$ )


Resources available  
( $A_1, A_2, A_3, \dots, A_m$ )

Current allocation matrix

Request matrix


$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation  
to process n


$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

恒等式: 
$$\sum_{i=1}^n C_{ij} + A_j = E_j$$



# 资源向量(矩阵)算法

- 每类资源多个的死锁检测算法
- 1.寻找进程 $P_i$ ，其在R矩阵中对应的第i行小于等于A
- 2. 如果找到，将C矩阵的第i行加入A，标记该进程执行完毕，转到第1步。
- 3.如果找不到，结束。
- 算法结束时，如存在未标记进程，则他们为死锁进程。

# 资源向量(矩阵)算法

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Tape drives  
Plotters  
Scanners  
Blu-rays

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

Tape drives  
Plotters  
Scanners  
Blu-rays

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

进程3可满足,  $A = (2 \ 2 \ 2 \ 0)$ ; 进程2可满足,  $A = (4 \ 2 \ 2 \ 1)$   
最后进程1 可满足

# 死锁恢复(deadlock recovery)

## ■ 资源抢占法

- 挂起一些占有资源的进程，剥夺它们的资源以解除死锁，将资源分配给另一个死锁进程使其能够执行完毕，然后再激活被挂起的进程。

## ■ 杀死进程法

- 杀死一个或者若干进程，释放其资源，直到打破死循环。
- 根据资源占有情况，杀死环内进或者环外进程
- 杀死重新执行无副作用的进程
  - 编译进程: OK
  - 数据库进程: ?
  - 打印进程: ?

# 死锁恢复(deadlock recovery)

## ■ 回滚法

- 设置检查点，根据死锁时所需要的资源，将一个拥有资源的进程滚回到一个未占用资源的检查点状态，从而使其他死锁进程能够获得相应的资源。
- 定期创建检查点
  - 保存存储镜像和资源获取的状态
  - 需要占用存储资源
- 检测到死锁后
  - 恢复到资源未分配时的检查点
  - 将资源分配给其他进程，继续执行

## ■ 回滚常常用来容错

- 数据库事务执行出错
- 高可用服务系统的容错-脱敏疗法

# 内容提要

- 死锁的概念
- 处理死锁的基本方法
  - 鸵鸟算法
  - 死锁检测
  - 死锁预防
  - 死所避免
- 小结

# 死锁预防

- 死锁预防（静态 deadlock avoidance）：破坏死锁的产生的四个条件之一。
  - 互斥、保持请求、不可抢占、环路等待

# 死锁预防 (1/4)

1. **打破互斥条件**：即允许进程同时访问某些资源。但是，有的资源是不允许被同时访问的，像打印机等等，这是资源本身的属性。

a) 使用假脱机技术可以将独占资源共享

2. **打破保持和请求条件**：在进程开始执行前请求所需的全部资源。只有当系统能够满足当前进程的全部资源需求时，才一次性地将所申请的资源全部分配给该进程，否则不分配任何资源。

a) 或者请求资源之前先释放，再尝试获得所有资源

# 死锁预防 (2/4)

但是，这种策略也有如下缺点：

- a) 在许多情况下，由于进程在执行时是动态的，**不可预测**的，因此不可能知道它所需要的全部资源。
- b) **资源利用率低**。无论资源何时用到，一个进程只有在占有所需的全部资源后才能执行。即使有些资源最后才被用到一次，但该进程在生存期间却一直占有。这显然是一种极大的资源浪费；
- c) **降低进程的并发性**。因为资源有限，又加上存在浪费，能分配到所需全部资源的进程个数就必然少了。



# 死锁预防 (3/4)

3. **打破不可抢占条件**：即允许进程强行从占有者那里夺取某些资源。由于资源的独占特性，这种预防死锁的方法实现起来困难。

a) 例如，把正在打印输出进程的打印机抢占，可能会导致打印异常。

# 死锁预防 (4/4)

4. **打破循环等待条件**：实行资源有序分配策略。即把资源事先分类编号，所有进程对资源的请求必须严格按资源序号递增的顺序提出，使进程在申请，占用资源时不会形成环路，从而预防了死锁。这种策略与前面的策略相比，资源的利用率和系统吞吐量都有很大提高，但存在以下缺点：
- a) 限制了进程对资源的请求，同时给系统中所有资源合理编号也是件困难事，并**增加了系统开销**；
  - b) 为了遵循按编号申请的次序，暂不使用的资源也需要提前申请，从而**增加了进程对资源的占用时间**。

# 有序资源分配法示例

例如：进程PA，使用资源的顺序是R1， R2；

进程PB，使用资源的顺序是R2， R1。

采用有序资源分配法：R1的编号为1， R2的编号为2；

PA：申请次序应是：R1， R2；

PB：申请次序应是：R1， R2。

# 哲学家进餐问题(the dining philosophers problem)

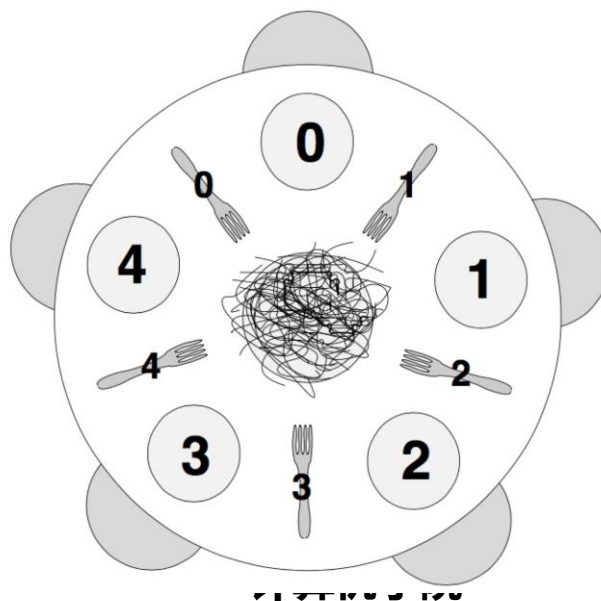
- Dijkstra, 1965.
- 5个哲学家围绕一张圆桌而坐，桌子上放着5支筷子(叉子)，每两个哲学家之间放一支；哲学家必须拿到左右两只筷子才能吃饭。

哲学家的循环动作：

```
while True:  
    think ()  
    get_forks()  
    eat()  
  
    put_forks()
```

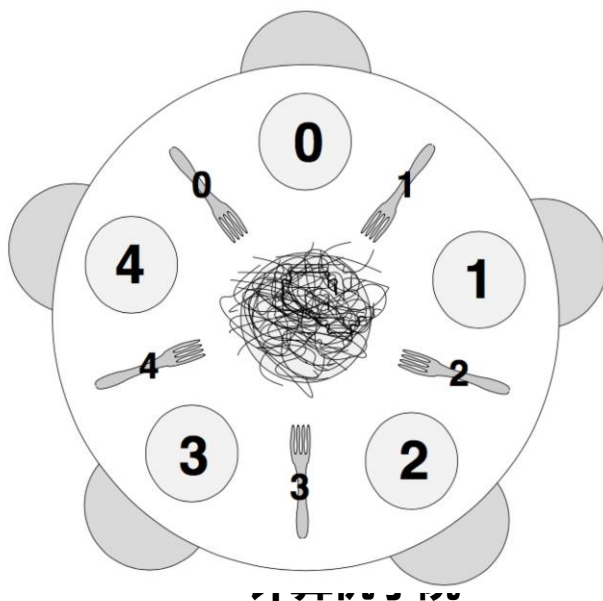
# 哲学家进餐问题(the dining philosophers problem)

- 5个哲学家围绕一张圆桌而坐，桌子上放着5支筷子(叉子)，每两个哲学家之间放一支；哲学家必须拿到左右两只筷子才能吃饭。
- 假设哲学家用 $i=0\sim 4$ 编号，叉子（筷子）也是。
- 哲学家 $i$ 必须拿到叉子(筷子) $i$ 和 $i+1$ 才能进食



# 哲学家进餐问题(the dining philosophers problem)

- 需要满足条件：
  - 假设一个哲学家一次只能拿到一个筷子
  - 不能死锁
  - 不能饿死
  - 不能只有一个哲学家进食（保证并发度）



# 哲学家进餐问题(the dining philosophers problem)

叉子的定义:

```
def left(i): return i
```

```
def right(i): return (i + 1) % 5
```

为每个叉子初始化一个信号量，一共5个。

```
forks = [Semaphore(i) for i in range(5)]
```

# 哲学家进餐问题(the dining philosophers problem)

```
def get_forks(i):  
    fork[right(i)].wait()  
    fork[left(i)].wait()  
  
def put_forks(i):  
    fork[right(i)].signal()  
    fork[left(i)].signal()
```

哲学家的循环动作:

```
while True:  
    think ()  
    get_forks()  
    eat()  
  
    put_forks()
```



# 哲学家进餐问题(the dining philosophers problem)

```
def get_forks(i):  
    fork[right(i)].wait()  
    fork[left(i)].wait()  
  
def put_forks(i):  
    fork[right(i)].signal()  
    fork[left(i)].signal()
```

哲学家的循环动作:

```
while True:  
    think ()  
    get_forks()  
    eat()  
  
    put_forks()
```

同时拿起右边的叉子，等待左边叉子，发生死锁！

# PV操作的描述

```
semaphore chopstick[5];
```

```
P(chopstick[i]);
```

```
P(chopstick[(i+1)mod 5]);
```

```
eat
```

```
V(chopstick[i]);
```

```
V(chopstick [(i+1)mod 5]);
```

```
think
```

同时拿起右边的叉子，等待左边叉子，发生死锁！

# 哲学家进餐问题(the dining philosophers problem)

```
def get_forks(i):  
    fork[right(i)].wait()  
    fork[left(i)].wait()  
  
def put_forks(i):  
    fork[right(i)].signal()  
    fork[left(i)].signal()
```

哲学家的循环动作:

```
while True:  
    think ()  
    get_forks()  
    eat()  
    put_forks()
```

如何通过破坏死锁的条件解决?

# 哲学家就餐问题的解题思路

- 至多只允许四个哲学家同时（尝试）进餐,以保证至少有一个哲学家能够进餐,最终总会释放出他所使用过的两支筷子,从而可使更多的哲学家进餐。设置信号量 `diners=4`（multiplex）。（破除循环等待）

```
Semaphore dinners=4;
```

```
def get_forks(i):  
    diners.wait()  
    fork[right(i)].wait()  
    fork[left(i)].wait()
```

```
def put_forks(i):  
    fork[right(i)].signal()  
    fork[left(i)].signal()
```

```
diners.signal()
```

# 哲学家就餐问题的解题思路

- 至多只允许四个哲学家同时（尝试）进餐,以保证至少有一个哲学家能够进餐 最终总会释放出他所使用过的两支

破除了死锁，避免了饿死。

```
semaphore dinners(4);  
def get_forks(i):  
    diners.wait()  
    fork[right(i)].wait()  
    fork[left(i)].wait()  
  
def put_forks(i):  
    fork[right(i)].signal()  
    fork[left(i)].signal()  
  
    diners.signal()
```

# 哲学家就餐问题的解题思路

- 假如至少一个左撇子和一个右撇子，则不会发生死锁。
  - 证明，反证法。
  - 假设死锁，那么五个哲学家都拿了一个叉子，等待另一个
  - 如果哲学家j左撇子，他一定左手拿叉等待右边叉子。
  - 那么其右边哲学家一定也是拿着左边叉子，等右边叉子，所以也是左撇子。
  - 如此推出，所有哲学家都是左撇子。
  - →和假设矛盾。

# 哲学家就餐问题的解题思路

- 对筷子进行编号，奇数号先拿左，再拿右；偶数号相反。  
（破除循环等待）

# 哲学家就餐问题的解题思路

- 同时拿起两根筷子，否则不拿起。（破除保持等待）



# 信号量集

放松了一次只能拿一个筷子的限制：（破除**保持等待**）

```
Var chopstick : array[0..4] of semaphore;
```

```
    think
```

```
    SP(chopstick[(i+1)mod 5], chopstick[i]);
```

```
    eat
```

```
    SV(chopstick[(i+1)mod 5], chopstick[i]);
```

# Tanebaum 算法

state = ['thinking'] \* 5

sem = [Semaphore(0) for i in range(5)]

mutex = Semaphore (1)

```
def get_fork(i):  
    mutex.wait()  
    state[i] = 'hungry'  
    test(i)  
    mutex.signal()  
    sem[i].wait()  
  
def put_fork(i):  
    mutex.wait()  
    state[i] = 'thinking'  
    test(right(i))  
    test(left(i))  
    mutex.signal()  
  
def test(i):  
    if state[i] == 'hungry' and  
       state[left (i)] != 'eating' and  
       state[right (i)] != 'eating':  
        state[i] = 'eating'  
        sem[i].signal()
```

# Tanebaum 算法

```
state = ['thinking'] * 5  
sem = [Semaphore(0) for i in range(5)]  
mutex = Semaphore(1)
```

```
def get_fork(i):  
    mutex.wait()
```

破除了死锁，但是仍然会导致饥饿。

Armando R. Gingras. Dining philosophers revisited.  
ACM SIGCSE Bulletin, 22(3):21–24, 28, September 1990.

```
def put_fork(i):  
    mutex.wait()  
    state[i] = 'thinking'  
    test(right(i))  
    test(left(i))  
    mutex.signal()  
  
def test(i):  
    if state[i] == 'hungry' and  
       state[left(i)] != 'eating' and  
       state[right(i)] != 'eating':  
        state[i] = 'eating'  
        sem[i].signal()
```

# 哲学家就餐问题的解题思路

- 对筷子进行编号，哲学家按照编号从低到高拿筷子。或者对哲学家编号，奇数号先拿左，再拿右；偶数号相反。（破除循环等待）
- 同时拿起两根筷子，否则不拿起。（破除保持等待）
- 至多只允许四个哲学家同时（尝试）进餐,以保证至少有一个哲学家能够进餐,最终总会释放出他所使用过的两支筷子,从而可使更多的哲学家进餐。设置信号量room = 4。（破除循环等待）

# 内容提要

- 死锁的概念
- 处理死锁的基本方法
  - 鸵鸟算法
  - 死锁检测
  - 死锁预防
  - 死锁避免
- 小结

# 死锁 (deadlock) -2

教师：姜博

E-Mail: [gongbell@gmail.com](mailto:gongbell@gmail.com)

# 内容提要

- 死锁的概念
- 处理死锁的基本方法
  - 鸵鸟算法
  - 死锁检测
  - 死锁预防
  - 死锁避免
- 小结

# 死锁避免 (Deadlock Avoidance)

- 死锁预防是排除死锁的静态策略，它使产生死锁的四个必要条件不能同时具备，从而对进程申请资源的活动加以限制，以保证死锁不会发生。
- 死锁的避免是排除死锁的动态策略，它不限制进程有关资源的申请，而是对进程所发出的每一个申请资源命令加以动态地检查，并根据检查结果决定是否进行资源分配。
  - 即分配资源时判断是否会出现死锁，有则加以避免。如不会死锁，则分配资源。
  - 假设(限制)：需要事先知道进程请求的所有资源



# 安全序列

- 安全序列的定义：所谓系统是安全的，是指系统中的所有进程能够按照某一种次序分配资源，并且依次地运行完毕，这种进程序列 $\{P_1, P_2, \dots, P_n\}$ 就是安全序列。
- 如果存在这样一个安全序列，则系统是安全的；如果系统不存在这样一个安全序列，则系统是不安全的。

# 安全状态

- **安全状态**：如果状态没有死锁发生，并且即使所有进程突然请求对资源的最大需求，也仍然存在某种调度次序能够使得每一个进程运行完毕，则称该状态是安全状态。
- **不安全状态**：不存在可完成的序列使进程运行完毕
- 系统进入不安全状态（四个死锁的必要条件同时发生）也未必会产生死锁。当然，产生死锁后，系统一定处于不安全状态。

进程	最大需求	已分配	可用
P1	10	5	3
P2	4	2	
P3	9	2	

<P2, P1, P3>      P3请求1

进程	最大需求	已分配	可用
P1	10	5	2
P2	4	2	
P3	9	3	

# 银行家算法

- 银行家算法 (Dijkstra, 1965)
  - 一个银行家把他的固定资金 (capital) 贷给若干顾客。只要不出现一个客户借走所有资金后还不够, 银行家的资金应是安全的。银行家需一个算法保证借出去的资金在有限时间内可收回。
  - 客户是进程, 银行家是操作系统, 贷款是满足进程的资源请求
- 为了保证资金的安全, 银行家规定:
  - 当一个客户对资金的最大需求量不超过银行家现有资金时就可接纳顾客 (分配资源)
  - 客户贷款总数不能超过最大需求量
  - 当银行家现有的资金不能满足客户尚需的贷款数额时, 对顾客的贷款可推迟支付, 但总能使客户在有限的时间里得到贷款 (等待其他进程释放资源再分配资源)
  - 当客户得到所需的全部资金后, 一定能在有限的时间里归还所有的资金 (资源必须释放)

# 具体算法

- 假定顾客分成若干次进行；并在第一次借款时，能说明他的最大借款额。具体算法：
  - 顾客的借款操作依次顺序进行，直到全部操作完成；
  - 银行家对当前顾客的借款操作进行判断，以确定其安全性（能否支持顾客借款，直到全部归还）；
  - 安全时，贷款；否则，暂不贷款。

- 可利用资源向量Available
- 最大需求矩阵Max
- 分配矩阵Allocation
- 需求矩阵Need

## ■ 可利用资源向量Available

- 一个含有 $m$ 个元素，其中每一个元素代表一类可利用的资源数目，其初值是系统中所配置的该类全部可用资源数目。如果 $Available[j]=k$ ，表示系统中现有 $R_j$ 类资源 $k$ 个。

## ■ 最大需求矩阵Max

- 一个含有 $n \times m$ 的矩阵，定义了系统中 $n$ 个进程中的每一个进程对 $m$ 类资源的最大需求。如果 $Max(i,j)=k$ ，表示进程 $i$ 需要 $R_j$ 类资源的最大数目为 $k$ 。



## ■ 分配矩阵Allocation

- 一个含有 $n \times m$ 的矩阵，定义了系统中每一类资源当前已分配给每一进程的资源数。如果 $\text{Allocation}(i,j)=k$ ，表示进程 $i$ 当前已分得 $R_j$ 类资源 $k$ 个。

## ■ 需求矩阵Need

- 一个含有 $n \times m$ 的矩阵，表示每一个进程尚需的各类资源数。如果 $\text{Need}(i,j)=k$ ，表示进程 $i$ 还需要 $R_j$ 类资源 $k$ 个，方能完成其任务。

$$\text{Need}(i,j) = \text{Max}(i,j) - \text{Allocation}(i,j)$$

# 银行家算法

设 $Request_i$ 是进程 $P_i$ 的请求向量，如果进程 $P_i$ 需要 $K$ 个 $R_j$ 类资源，当 $P_i$ 发出资源请求后，系统按下述步骤进行检查：

- 1 如果 $Request_i \leq Need_i$ ，则转向步骤2；否则认为出错。（因为它所需要的资源数已超过它所宣布的最大值。）
- 2 如果 $Request_i \leq Available$ ，则转向步骤3；否则，表示系统中尚无足够的资源， $P_i$ 必须等待
- 3 系统**试探**把要求的资源分配给进程 $P_i$ ，并修改下面数据结构中的数值：

$Available := Available - Request_i;$

$Allocation := Allocation + Request_i;$

$Need_i := Need_i - Request_i;$

- 4 **系统执行安全性算法**，检查此次资源分配后，系统是否处于安全状态。若安全，正式将资源分配给进程 $P_i$ ，以完成本次分配；否则，将试探分配作废，恢复原来的资源分配状态，让进程 $P_i$ 等待。

# 安全性算法

系统所执行的安全性算法可描述如下：

1 设置两个向量

- ① 工作向量Work. 它表示系统可提供给进程继续运行所需要的各类资源的数目，它含有  $m$  个元素，执行安全算法开始时， $Work := Available$ 。
  - ② Finish. 它表示系统是否有足够的资源分配给进程，使之运行完成。开始时先做  $Finish[i] := false$ ；当有足够的资源分配给进程  $i$  时，令  $Finish[i] := true$ 。
- 2 从进程集合中找到一个能满足下述条件的进程：①  $Finish[i] = false$ ；②  $Need_i \leq Work$ 。如找到，执行步骤3；否则执行步骤4。
- 3 当进程  $P_i$  获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故执行：
- $Work := Work + Allocation;$   
 $Finish[i] := true;$   
Goto step2;
- 4 如果所有进程的  $Finish[i] = true$ ，则表示系统处于安全状态；否则，系统处于不安全状态。

# 银行家算法示例

- 假定系统中有五个进程{P0、P1、P2、P3、P4}和三种类型的资源{A, B, C}, 每一种资源的数量分别为10、5、7, 在T0时刻的资源分配情况如下表所示:

资源情况 进程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	7	5	3	0	1	0	7	4	3	3	3	2
P1	3	2	2	2	0	0	1	2	2			
P2	9	0	2	3	0	2	6	0	0			
P3	2	2	2	2	1	1	0	1	1			
P4	4	3	3	0	0	2	4	3	1			

10,5 7

资源情况 进程		最大值	已分配	还需要	可用
		Max	Allocation	Need	Available
		A B C	A B C	A B C	A B C
	P0	7 5 3	0 1 0	7 4 3	3 3 2
	P1	3 2 2	2 0 0	1 2 2	
	P2	9 0 2	3 0 2	6 0 0	
	P3	2 2 2	2 1 1	0 1 1	
	P4	4 3 3	0 0 2	4 3 1	

工作向量Work.它表示系统可提供给进程继续运行所需要的各类资源的数目

资源情况 进程		work	Need	Allocation	Work + Allocation	finish
		A B C	A B C	A B C	A B C	
	P1	3 3 2	1 2 2	2 0 0	5 3 2	true
	P3	5 3 2	0 1 1	2 1 1	7 4 3	true
	P4	7 4 3	4 3 1	0 0 2	7 4 5	true
	P2	7 4 5	6 0 0	3 0 2	10 4 7	true
	P0	10 4 7	7 4 3	0 1 0	10 5 7	true

# 银行家算法的特点

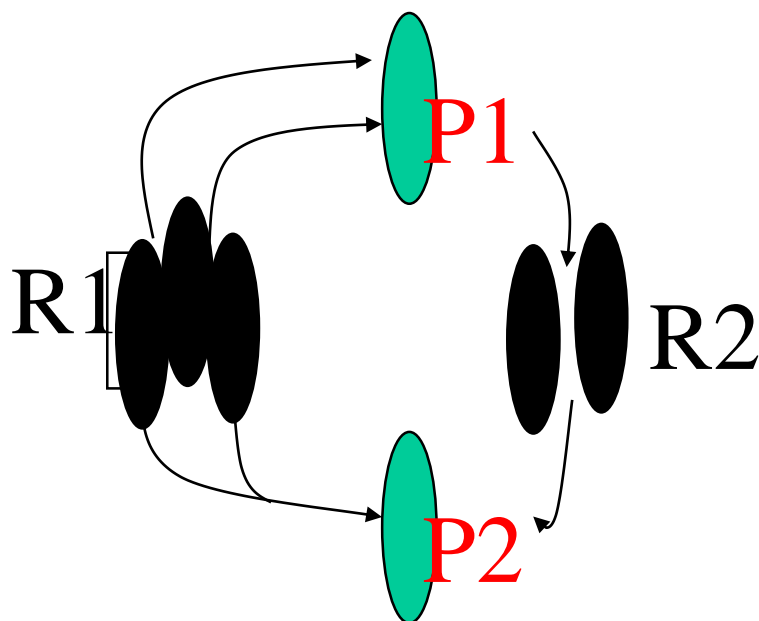
- 允许互斥、部分分配和不可抢占，可提高资源利用率；
- 要求事先说明最大资源要求，在现实中无法实用；

- **封锁进程**：是指某个进程由于请求了超过了系统中现有的未分配资源数目的资源，而被系统封锁的进程。
- **非封锁进程**：即没有被系统封锁的进程
- **资源分配图的化简方法**：假设某个RAG中存在一个进程 $P_i$ ，此刻 $P_i$ 是非封锁进程，那么可以进行如下化简：
  - 当 $P_i$ 有请求边时，首先将其请求边变成分配边(即满足 $P_i$ 的资源请求)，而一旦 $P_i$ 的所有资源请求都得到满足， $P_i$ 就能在有限的时间内运行结束，并释放其所占用的全部资源，此时 $P_i$ 只有分配边，删去这些分配边（实际上相当于消去了 $P_i$ 的所有请求边和分配边），使 $P_i$ 成为孤立结点。（反复进行）

## 死锁定理:

系统中某个时刻 $t$ 为死锁状态的充要条件是 $t$ 时刻系统的资源分配图是不可完全化简的。

在经过一系列的简化后，若能消去图中的所有边，使所有的进程都成为孤立结点，则称该图是可完全化简的；反之的是不可完全化简的。





# 思考

- 仅涉及一个进程的死锁有可能存在吗?为什么?

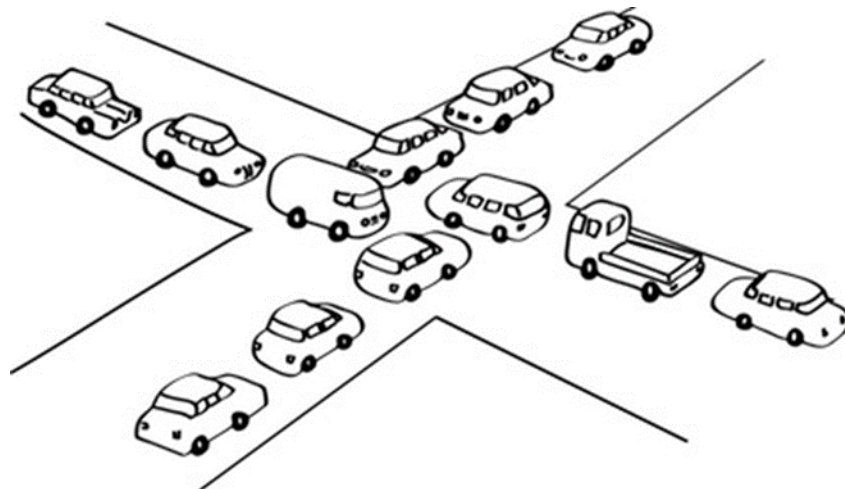
# 思考

- 仅涉及一个进程的死锁有可能存在吗?为什么?
- 环路等待条件: 两个或者多个进程组成环路。

# 如何用描述该死锁问题？



# 如何用描述该死锁问题？



假设是双向车道，路口可以分成四个方格，每个方格是一个资源 $R_1, R_2, R_3, R_4$ ；东西南北四个方向的车辆相当于四个进程 $P_1, P_2, P_3, P_4$ ；每个车辆要通过路口需要占用两个资源，当出现环路时，产生了死锁。

假定系统中有五个进程{P0、P1、P2、P3、P4}和三种类型的资源{A, B, C}，每一种资源的数量分别为10、5、7，在T0时刻的资源分配情况如图

请找出该表中T0时刻以后存在的安全序列（至少2种）

资源情况 进程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	7	5	3	0	1	0	7	4	3	3	3	2
P1	3	2	2	2	0	0	1	2	2			
P2	9	0	2	3	0	2	6	0	0			
P3	2	2	2	2	1	1	0	1	1			
P4	4	3	3	0	0	2	4	3	1			

- 如果P4请求分配(3,3,0)，是否可以？
- 如果P0请求分配 (0,2,0)，是否可以？

# 小结

- 死锁的概念
- 处理死锁的基本方法
  - 预防死锁
  - 避免死锁
  - 检测死锁
  - 解除死锁

谢谢!  
gongbell@gmail.com