

计算机系统

教师：王雷

82316284, wanglei@buaa.edu.cn

内容提要

- 死锁的概念
- 处理死锁的基本方法
- 小结

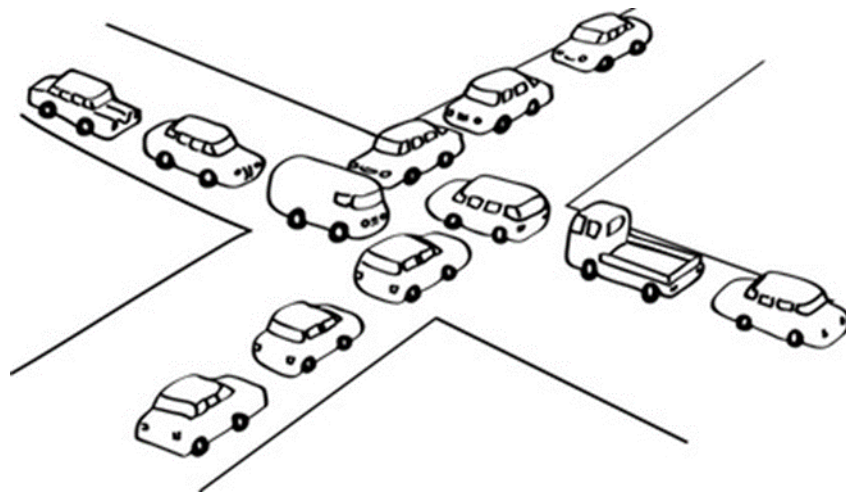
死锁问题(Deadlock)

死锁定义：

- 由于资源占用的互斥，当某个进程提出资源申请之后，使得一些进程在无外力协助的情况下，永远分配不到必需的资源而无法运行。

死锁发生原因

- 竞争资源
- 并发执行的顺序不当



死锁的例子

进程P1

...

申请文件F

申请打印机T

...

释放打印机T

释放文件F

...

进程P2

...

申请打印机T

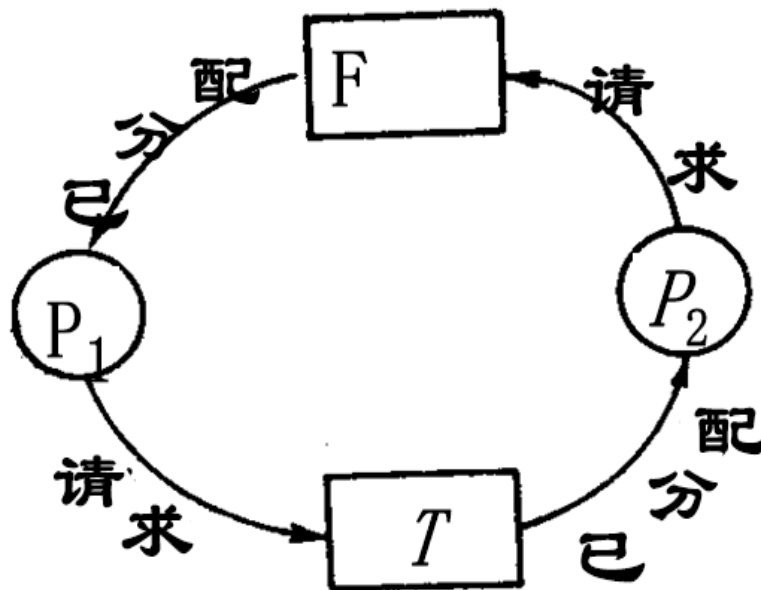
申请文件F

...

释放文件F

释放打印机T

...



竞争资源引起死锁

- **可剥夺资源**：是指某进程在获得这类资源后，该资源可以再被其他进程或系统剥夺。如**CPU**，**内存**；
- **非可剥夺资源**：当系统把这类资源分配给某进程后，再不能强行收回，只能在进程用完后自行释放。如**磁带机**、**打印机**；
- **临时性资源**：这是指由一个进程产生，被另一个进程使用，短时间后便无用的资源，故也称为消耗性资源。如**消息**、**中断**；

临时性资源竞争示例

例如，S1，S2，S3是临时性资源，进程P1产生消息S1，又要求从P3接收消息S3；进程P3产生消息S3，又要求从进程P2处接收消息S2；进程P2产生消息S2，又要求从P1处接收产生的消息S1。如果消息通信按如下顺序进行：

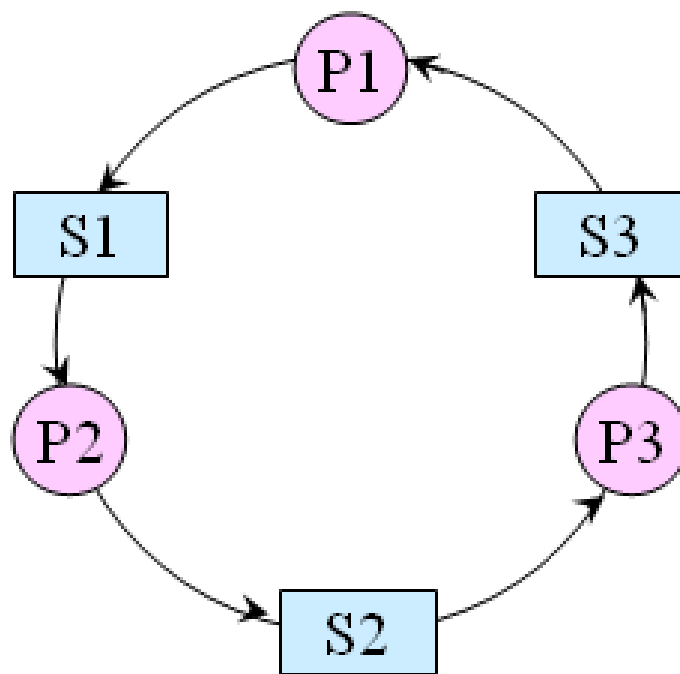
P1: Release (S1) ; Request (S3) ;
P2: Release (S2) ; Request (S1) ;
P3: Release (S3) ; Request (S2) ;

并不会发生死锁。

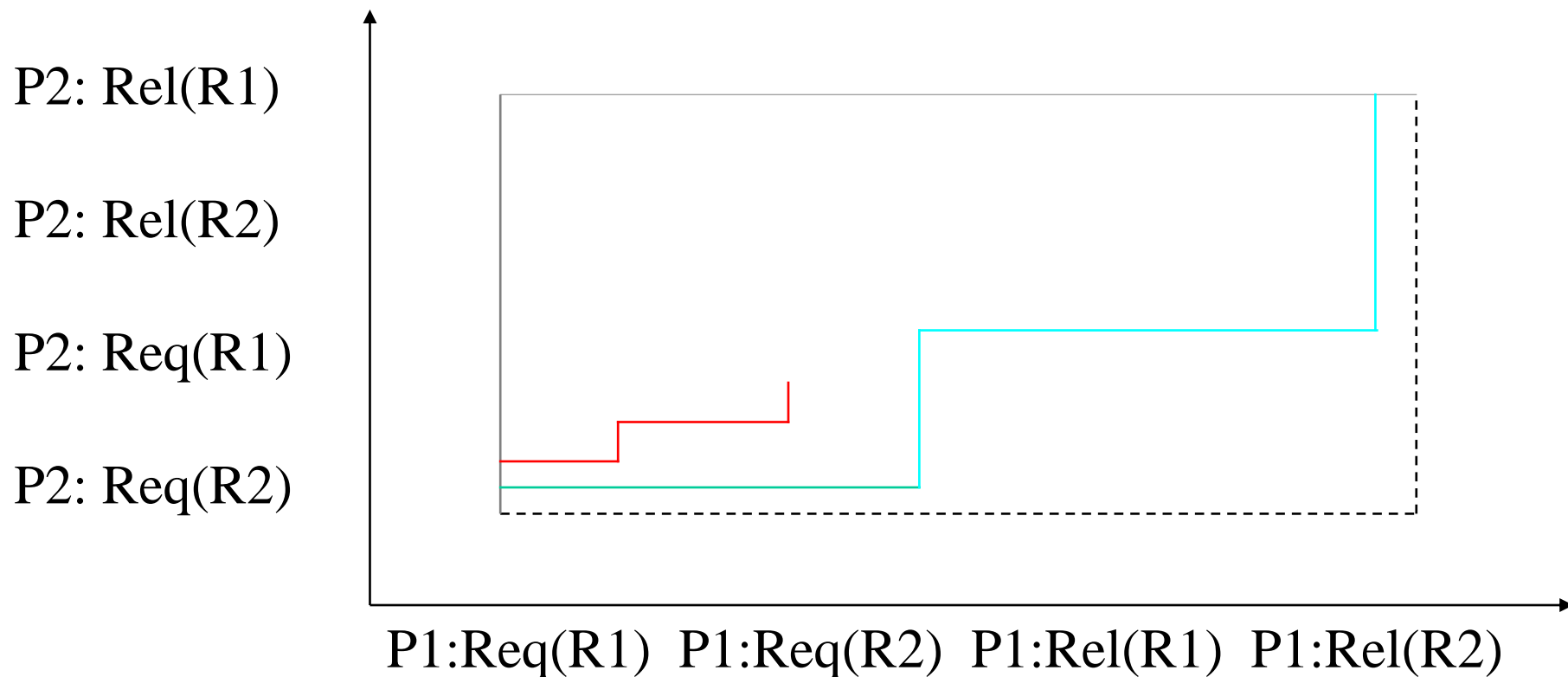
若改成下述的运行顺序：

P1: Request (S3) ; Release (S1) ;
P2: Request (S1) ; Release (S2) ;
P3: Request (S2) ; Release (S3) ;

则可能发生死锁。



进程推进顺序不当引起死锁



使用信号量实现会合 (Rendezvous)

- 使用信号量实现线程A和线程B的会合(Rendezvous)。使得a1永远在b2之前，而b1永远在a2之前。
- 定义两个信号量，aArrived, bArrived,并且初始化为0，表示a和b是否执行到汇合点。

Thread A

```
1 statement a1
2 bArrived.wait()
3 aArrived.signal()
4 statement a2
```

Thread B

```
1 statement b1
2 aArrived.wait()
3 bArrived.signal()
4 statement b2
```

死锁版本

- full是“满”数目，初值为0，empty是“空”数目，初值为N。实际上，full和empty是同一个含义： $full + empty == N$
- mutex用于访问缓冲区时的互斥，初值是1

生产者

消费者

P(mutex);

P(empty);

one >> buffer

V(full)

V(mutex)

P(mutex);

P(full);

one << buffer

V(empty)

V(mutex)



PV操作的描述

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

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！

死锁发生的四个必要条件

1. **互斥条件**：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。
2. **请求和保持条件**：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。
3. **不剥夺条件**：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。
4. **环路等待条件**：指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合 $\{P_0, P_1, P_2, \dots, P_n\}$ 中的 P_0 正在等待一个 P_1 占用的资源； P_1 正在等待 P_2 占用的资源，……， P_n 正在等待已被 P_0 占用的资源。

内容提要

- 死锁的概念
- 处理死锁的基本方法
- 小结

处理死锁方法

- 不允许死锁发生
 - 预防死锁（静态）：防患于未然，破坏死锁的产生条件
 - 避免死锁（动态）：在资源分配之前进行判断
- 允许死锁发生
 - 检测与解除死锁
 - 无所作为：鸵鸟算法

严格性

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。
2. **打破占有且申请条件**：可以实行资源预先分配策略。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。由于运行的进程已占有了它所需的全部资源，所以不会发生占有资源又申请资源的现象，因此不会发生死锁。

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. 打破不可剥夺条件：即允许进程强行从占有者那里夺取某些资源。就是说，当一个进程已占有了某些资源，它又申请新的资源，当不能立即被满足时，须释放所占有的全部资源，以后再重新申请。它所释放的资源可以分配给其它进程。这就相当于该进程占有的资源被隐蔽地强占了。这种预防死锁的方法实现起来困难，会降低系统性能。

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，按号分配，使进程在申请，占用资源时不会形成环路。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。进程占用了小号资源，才能申请大号资源，就不会产生环路，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)

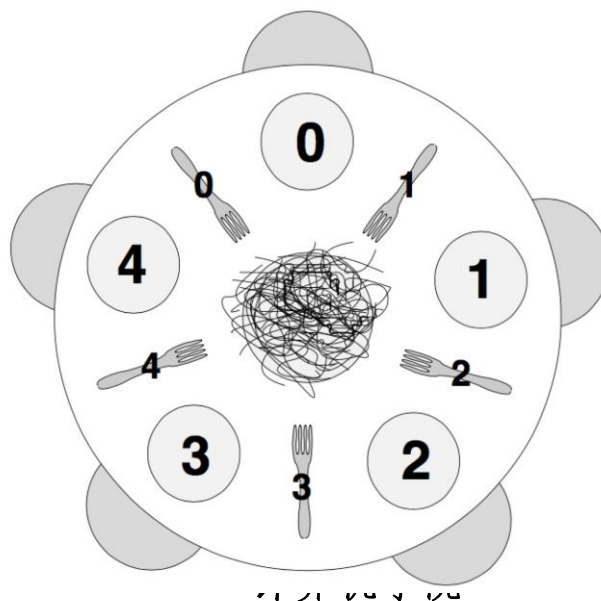
- Dijkstra, 1965.
- 5个哲学家围绕一张圆桌而坐，桌子上放着5支筷子(叉子)，每两个哲学家之间放一支；哲学家必须拿到左右两只筷子才能吃饭。

哲学家的循环动作：

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

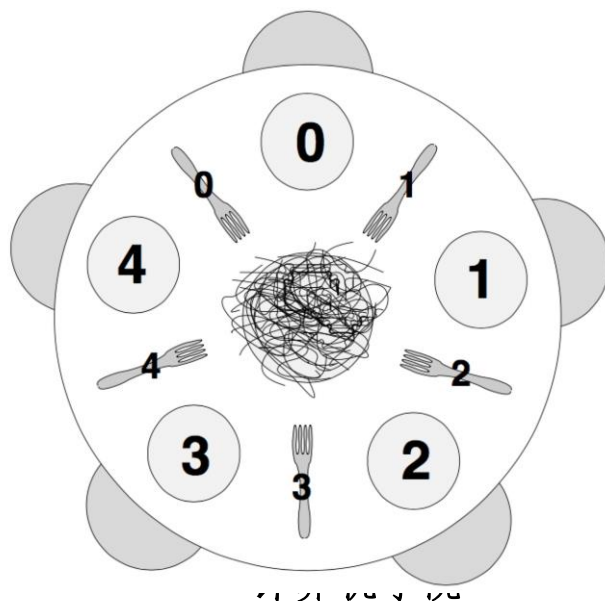
哲学家进餐问题(the dining philosophers problem)

- 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！

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]);
```

避免死锁

- **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。
- 死锁避免不那么严格限制产生死锁的四个必要条件（区别于死锁预防）

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。
- 安全序列 $\{P_1, P_2, \dots, P_n\}$ 是这样组成的：若对于每一个进程 P_i ，它需要的附加资源可以被系统中当前可用资源加上所有进程 P_j 当前占有资源之和所满足，则 $\{P_1, P_2, \dots, P_n\}$ 为一个安全序列。

安全状态

- 安全状态：系统存在一个进程执行序列 $\langle p_1, p_2, \dots, p_n \rangle$ 可顺利完成
- 不安全状态：不存在可完成的序列
- 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。



安全状态示例

已有数量 最大需求

A	3	9
B	2	4
C	2	7

空闲：3

A	3	9
B	4	4
C	2	7

空闲：1

A	3	9
B	0	—
C	2	7

空闲：5

A	3	9
B	0	—
C	7	7

空闲：0

A	3	9
B	0	—
C	0	—

空闲：7

安全

已有数量 最大需求

A	3	9
B	2	4
C	2	7

空闲：3

A	4	9
B	2	4
C	2	7

空闲：2

A	4	9
B	4	4
C	2	7

空闲：0

A	4	9
B	0	—
C	2	7

空闲：4

不安全

原因：为进程A分配了资源导致

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。只要不出现一个顾客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。

- 为了保证资金的安全, 银行家规定:

- 当一个顾客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
- 顾客可以分期贷款, 但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顾客尚需的贷款数额时, 对顾客的贷款可推迟支付, 但总能使顾客在有限的时间里得到贷款
- 当顾客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

具体算法

- 假定顾客借款分成若干次进行；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。

具体算法

- **n**为进程数量，**m**为资源类型数量
- 可利用资源向量**Available**: **m**维向量
 - 具有**m**个元素的向量，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**: **n**×**m**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**。

具体算法

- 分配矩阵**Allocation**: $n \times m$ 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**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**: $n \times m$ 矩阵
 - 表示每一个进程尚需的各类资源数。如果**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， $Work := Available$ 。

②Finish. 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

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？

银行家算法的特点

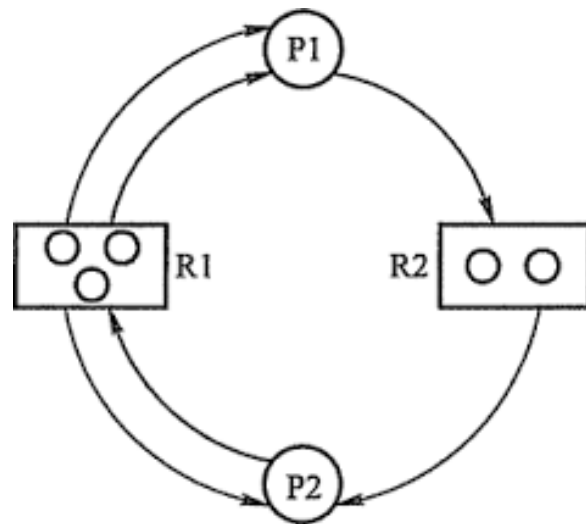
- 允许互斥、部分分配和不可抢占，可提高资源利用率；
- 要求事先说明最大资源要求，在现实中很困难；

检测死锁

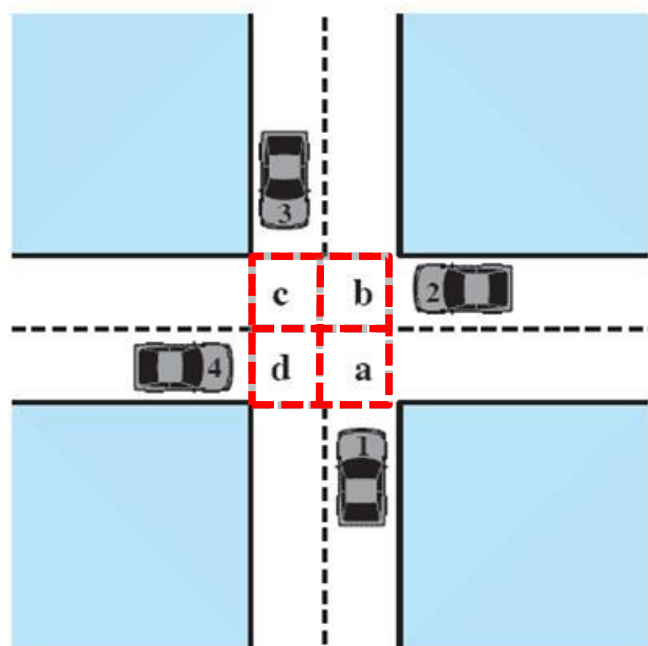
- 保存资源的请求和分配信息，利用某种算法对这些信息加以检查，以判断是否存在死锁。死锁检测算法主要是检查是否有循环等待。
- 在UNIX系统中，PS；Windows中的任务管理器

资源分配图/进程-资源图

- 用有向图描述系统资源和进程的状态。二元组 $G = (N, E)$ ， N ：结点的集合， $N = P \cup R$ 。
- P 为进程， R 为资源， $P = \{p_1, p_2, \dots, p_n\}$ ， $R = \{r_1, r_2, \dots, r_m\}$ ，两者为互斥资源。
- E ：有向边的集合， $e \in E$ ， $e = (p_i, r_j)$ 或 $e = (r_j, p_i)$ 。
 - $e = (p_i, r_j)$ 是请求边，进程 p_i 请求一个单位的 r_j 资源；
 - $e = (r_j, p_i)$ 是分配边，为进程 p_i 分配了一个单位的 r_j 资源。
- 在资源分配图中，圆圈表示进程，矩形表示一类资源，矩形中的小圈代表每个资源。



如何用描述该死锁问题？

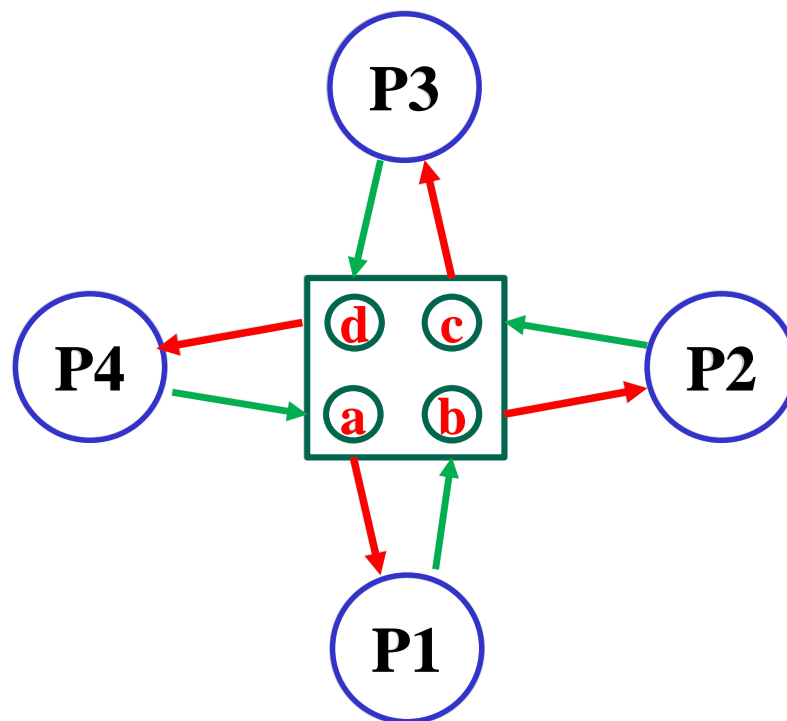
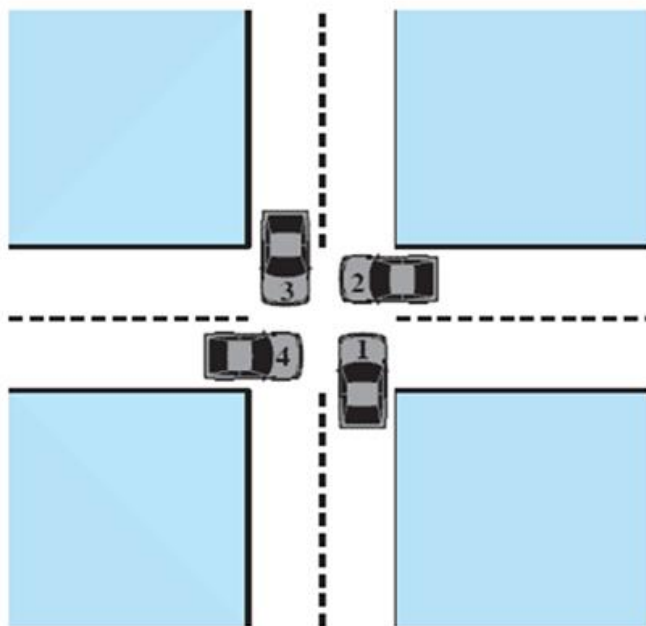


假设是双向车道：

- 路口可以分成四个方格，每个方格是一个资源 **Ra, Rb, Rc, Rd**;
- 东西南北四个方向的车辆相当于四个进程 **P1, P2, P3, P4**;
- 每个车辆要通过路口需要占用两个资源。

如何用描述该死锁问题？

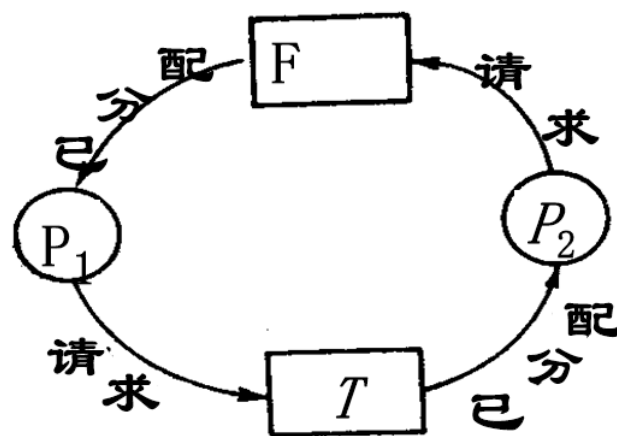
- 当每辆车都驶入路口，相当于占用了—个资源，出现环路时，产生了死锁。



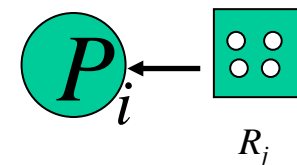
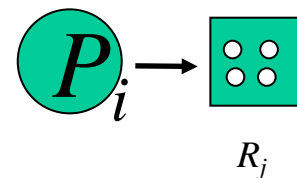
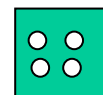
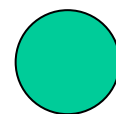
资源分配图 (RAG) 算法

- RAG (Resource Allocation Graph)
- 有向图G的顶点为资源或进程，从资源R到进程P的边表示R已分配给P，从进程P到资源R的边表示P正因请求R而处于等待状态。

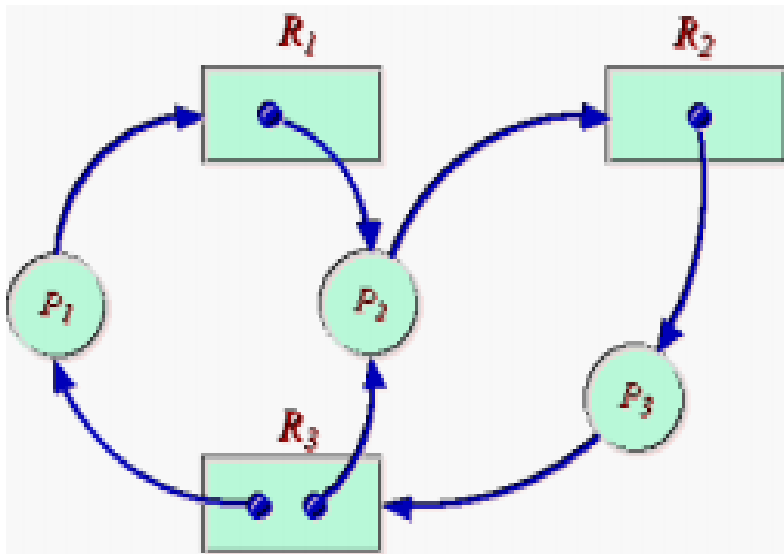
思考：如果一个资源分配图中存在环路，是否一定存在死锁？



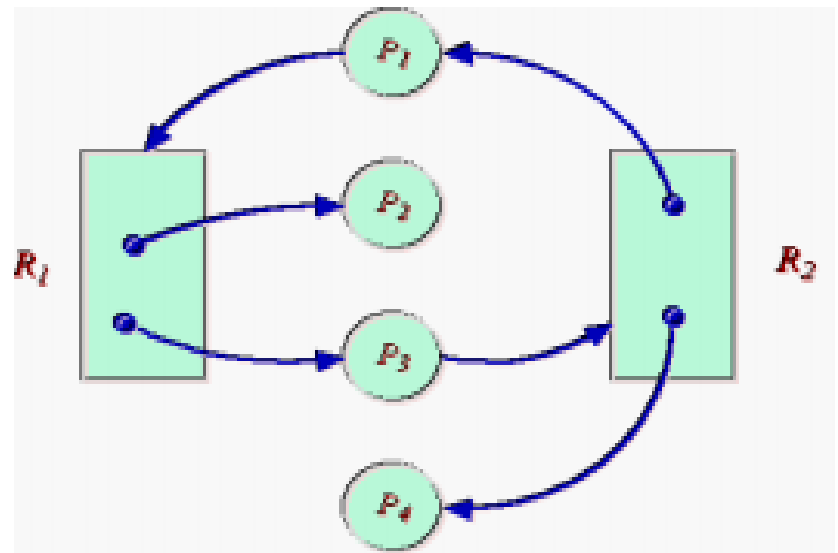
简单的死锁例子



“环”与死锁



有环有死锁



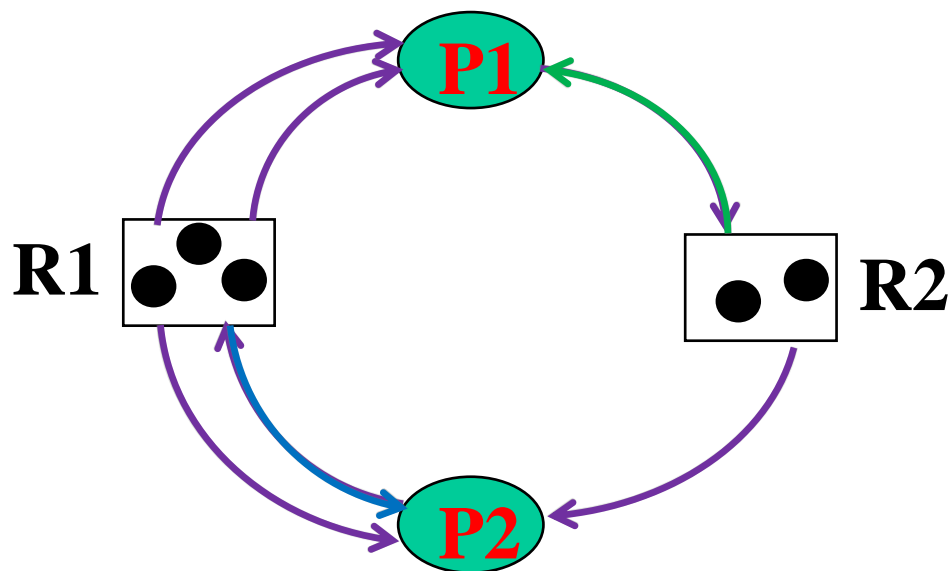
有环无死锁

- **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。



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。

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。
- 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

- 实质：如何让释放资源的进程能够继续运行
 - 选择一个牺牲进程
 - 重新运行或回退到某一点开始继续运行
 - 回退到足以解除死锁即可
- 须注意的问题：
 - 怎样保证不发生“饥饿”现象，如何保证并不总是剥夺同一进程的资源
 - “最小代价”，即最经济合算的算法，使得进程回退带来的开销最小。

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

死锁检测、预防和避免方法小结

原则	资源分配策略	不同方案	主要优点	主要缺点
预防	保守的；预提交资源，导致资源闲置	一次性请求所有资源	<ul style="list-style-type: none"> 对执行一连串活动（突发式处理）的进程非常有利 不需要抢占 	<ul style="list-style-type: none"> 低效，延迟进程的初始化 须知道未来的资源情况 资源利用率低
		抢占	对状态易于保存和恢复的资源非常方便	<ul style="list-style-type: none"> 可能导致过于频繁的抢占
		资源排序	可在系统设计时解决，在编译时实施。	<ul style="list-style-type: none"> 不便灵活申请新资源
避免	是“预防”与“检测”的折衷	通过资源需求检查以发现至少一条安全路径	不需要抢占	<ul style="list-style-type: none"> 须知道未来资源的需求， 进程可能被长时间阻塞
检测	宽松的，只要可能，请求的资源都允许	周期性地调用以检测死锁	<ul style="list-style-type: none"> 不会延迟进程的初始化 易于在线处理 	<ul style="list-style-type: none"> 通过抢占解除死锁，可能造成损失

生产者/消费者问题—sleep&wakeup

```
#define N 100
```

```
int count=0;
```

```
void producer(void)
```

```
{ int item;
```

```
while(TRUE) {
```

```
    item=produce_item();
```

```
    if(count==N) sleep();
```

```
    insert_item(item);
```

```
    count=count+1;
```

```
    if(count==1)
```

```
        wakeup(consumer);
```

```
}
```

```
}
```

检查
count
的值

```
void consumer(void)
```

```
{
```

```
    int item;
```

```
    while(TRUE) {
```

```
        if(count==0) sleep();
```

```
        item=remove_item();
```

```
        count=count-1;
```

```
        if(count==N-1)
```

```
            wakeup(producer);
```

```
        consume_item(item);
```

```
    }
```

```
}
```

一种场景：消费者
判断count=0后进入
睡眠前被切换

Sleep&wakeup算法的问题

- 对count的读取和睡眠应该是不能中断的原子操作!
- 缓冲区为空→消费者读取count发现为0→消费者被切换→生产者被调度→生产者加入数据，count=1→推断消费者在睡眠，wakeup消费者
- 但是消费者只是被切换，没有睡眠→wakeup信号丢失
- 消费者被调度→发现之前读取的count为0→睡眠
- 生产者被调度→不断填满缓冲区→睡眠
- 全都睡眠!

通信死锁!

活锁和饥饿

- **活锁 (livelock)**：是指任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。
 - 活锁和死锁的区别在于，处于活锁的实体是在不断的改变状态，即所谓的“活”，而处于死锁的实体表现为等待；活锁有可能自行解开，死锁则不能。避免活锁的简单方法是采用先来先服务的策略。
- **饥饿 (starvation)**：某些进程可能由于资源分配策略的不公平导致长时间等待。当等待时间给进程推进和响应带来明显影响时，称发生了进程饥饿，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(starve to death)。

思考

- 1.在某系统中，三个进程共享四台同类型的设备资源，这些资源一次只能一台一台地为进程服务和释放，每个进程最多需要二台设备资源，试问在系统中是否会产生死锁？
- 2.某系统中有 n 个进程和 m 台打印机，系统约定：打印机只能一台一台地申请、一台一台地释放，每个进程需要同时使用的打印机台数不超过 m 。如果 n 个进程同时需要使用打印机的总数小于 $m+n$ ，试讨论，该系统可能发生死锁吗？并简述理由。
- 3.仅涉及一个进程的死锁有可能存在吗？为什么？

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