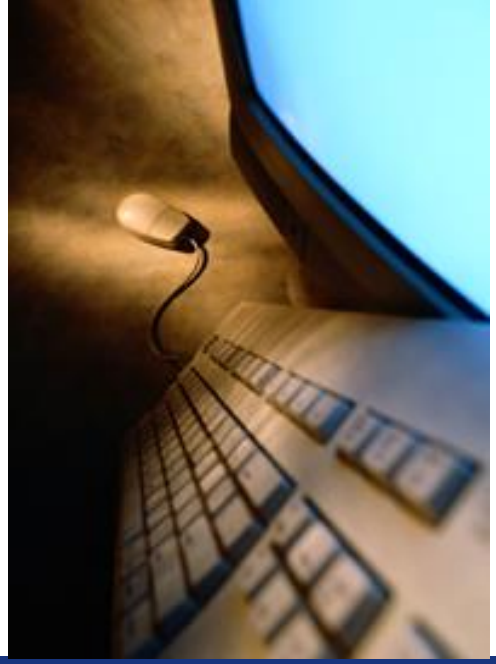
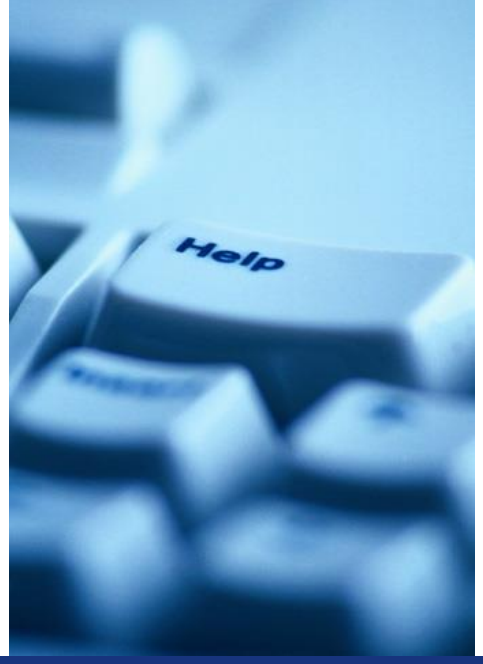



操作系统原理

Operating Systems Principles

张青
计算机学院

第八讲 —并发: 死锁、 饥饿

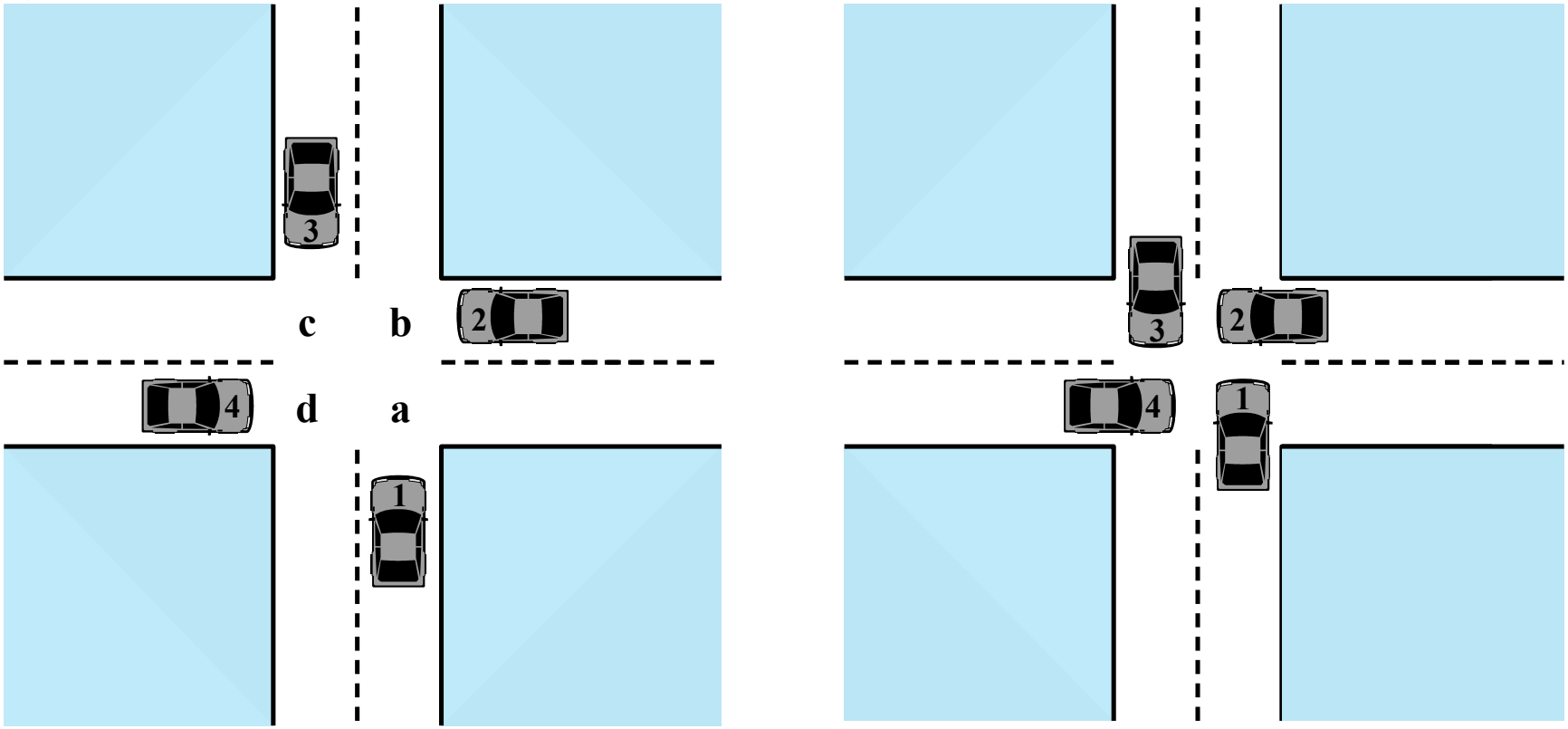
目标

- 掌握死锁产生的条件；
- 定义死锁预防、提出死锁预防的策略；
- 理解死锁预防与死锁避免的区别；
- 掌握死锁避免的两种方法；
- 理解死锁检测与死锁预防；
- 掌握设计综合死锁解决策略；
- 分析哲学家就餐问题；
- 理解UNIX、Linux等操作系统中的并发和同步机制；

死锁

- 当一组进程中的每个进程都在等待某个事件（资源），而仅有这组进程中被阻塞的其他进程才可触发该事件时，则认为该组进程发生了死锁。
- 死锁是永久性的；
- 死锁问题并没有有效而通用的解决方案；

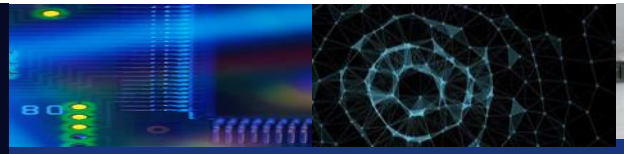
死锁



(a) Deadlock possible　　(b) Deadlock

所有的死锁都涉及两个或者多个进程之间对资源需求的冲突

Figure 6.1　Illustration of Deadlock



资源分类

Reusable

- can be safely used by only one process at a time and is not depleted by that use
  - processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores

进程使用资源的正确顺序：申请、使用、释放

Consumable

- one that can be created (produced) and destroyed (consumed)
  - interrupts, signals, messages, and information
  - in I/O buffers

进程竞争可重用资源

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q

Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

Figure 6.4
Example of Two Processes Competing for Reusable Resources

可消耗资源

- 考虑下面的进程对，其中每个进程都试图从另一个进程接收消息，然后再给那个进程发送一条消息；

P1	P2
...	...
Receive (P2);	Receive (P1);
...	...
Send (P2, M1);	Send (P1, M2);

- **Receive**阻塞时发生死锁，程序可能运行一段比较长的时间后才会被发现；

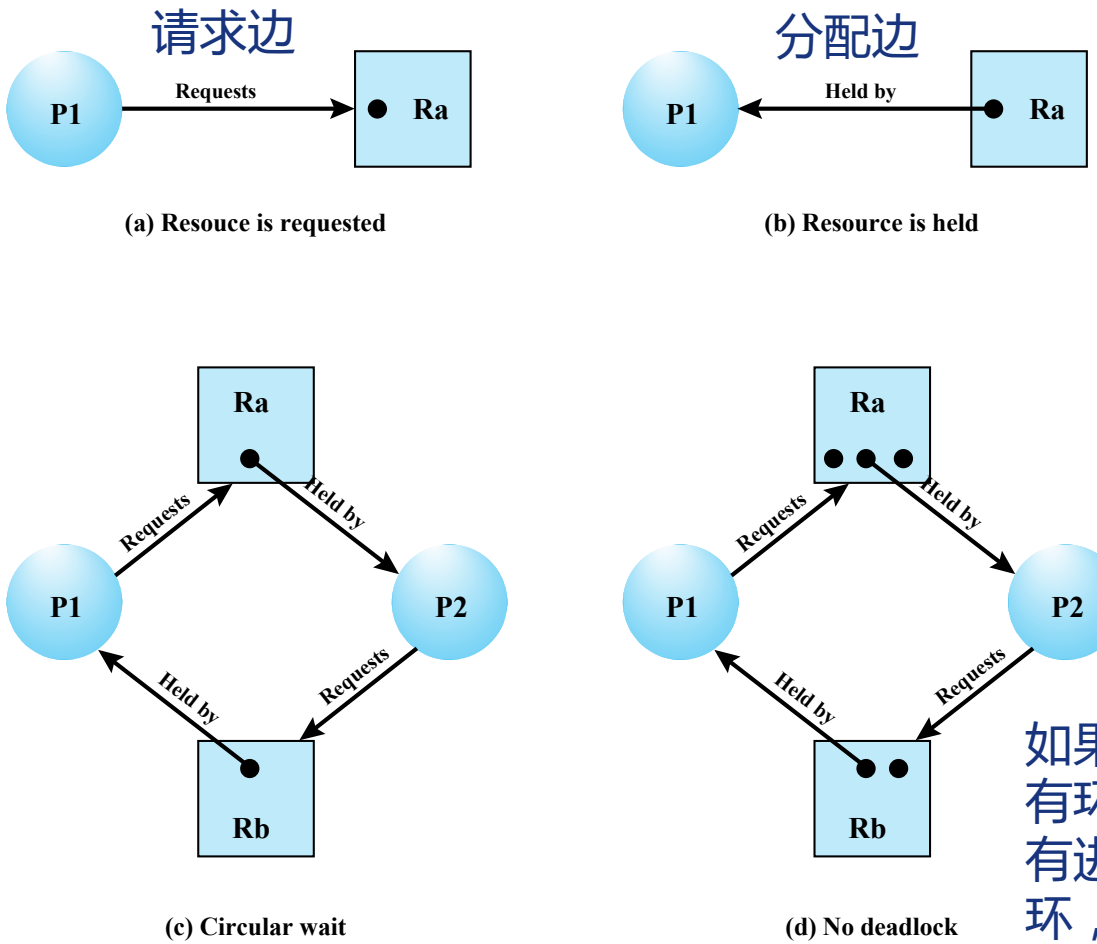
死锁的必要条件

死锁的存在性

互斥	占有等待	非抢占	循环等待
• 一次只有一个进程使用一个资源，其他进程不能访问分配给其他进程的资源；	• 当一个进程等待其他进程时，继续占有已分配的资源；	• 不能强行抢占进程已占有的资源；	• 存在一个闭合的进程链，每个进程至少占有此链中下一个进程所需的一个资源；

死锁的可能性

资源分配图

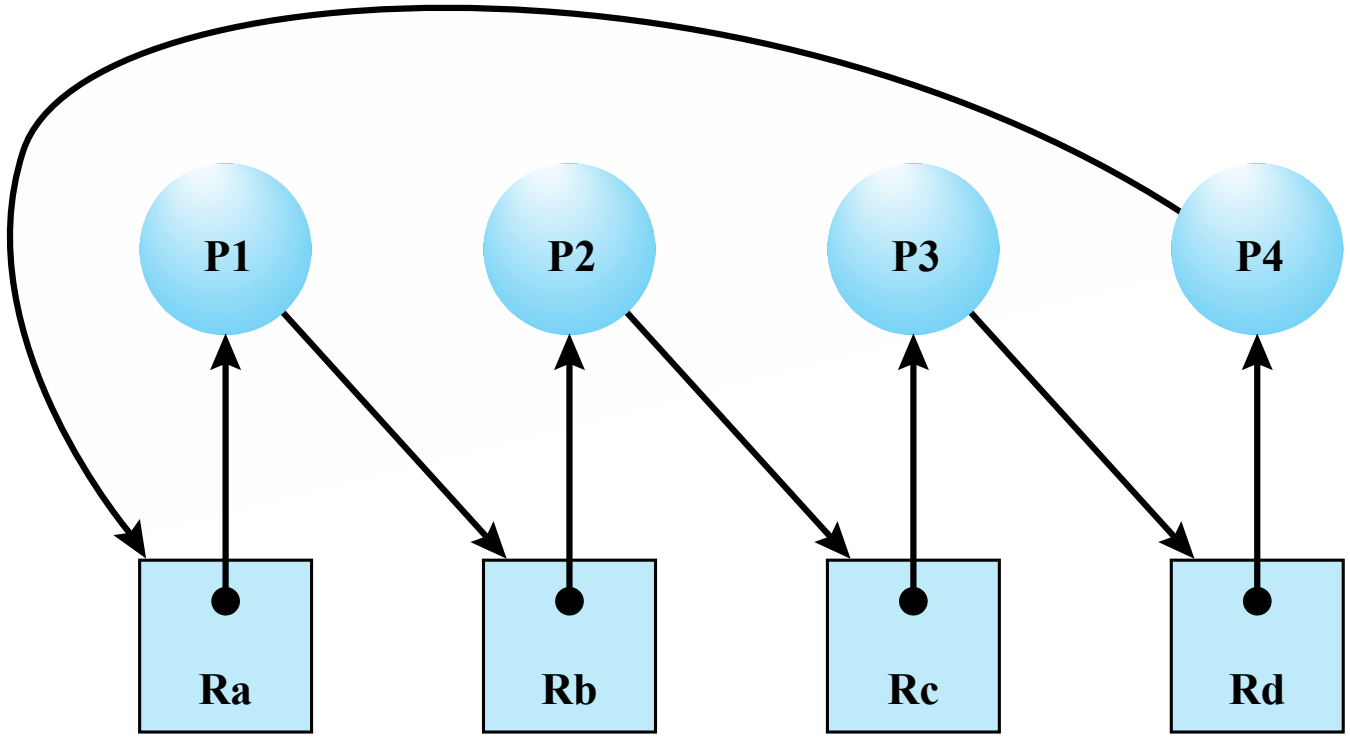


如果资源分配图中没有环，那么系统就没有进程死锁，如果有环，可能存在死锁

表征进程资源分配的有效工具是资源分配图

Figure 6.5 Examples of Resource Allocation Graphs

资源分配图



导致死锁的资源分配情况

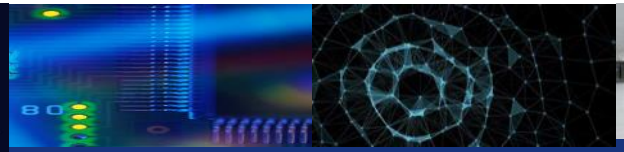
处理死锁的方法

- **确保系统永远不会进入死锁状态：**
  - 死锁预防：确保至少有一个死锁必要条件不成立
  - 死锁避免：操作系统事先得到有关进程申请资源和使用资源的额外信息并基于此来判断进程是否应等待
- **允许系统进入死锁状态，然后检测和恢复；**
- **鸵鸟策略：忽略该问题，并假装系统中从未发生死锁；**
  - **绝大多数操作系统，包括Linux、Unix和Windows，所采用的方法；**

为什么可以采用鸵鸟策略？

✓ 解决死锁的代价很高，采用鸵鸟策略：不采取任何措施，能获得更高的性能。

✓ 死锁发生的概率很低，就算发生死锁对用户的影响并不大，所以可以采用鸵鸟策略。



死锁预防策略

使死锁的四个必要条件之一无效：

- **互斥-可共享资源（例如只读文件）不需要互斥；通常不能否定互斥条件来预防死锁，必须保留不可共享的资源；**
- **占有等待 - 必须保证每当进程请求资源时，它不会保持任何其他资源；**
  - 一种实现：在开始执行之前请求并分配其所有资源;另一种实现：仅当进程在未分配任何资源时才允许进程请求资源。
  - 上述两种方法缺点是资源利用率可能比较低，存在资源已经分配，但是长时间不用；发生饥饿，长久等待需要的资源（尤其是需要多个多个常用资源）；

死锁预防策略

- **无抢占：**
  - 如果持有某些资源的进程请求另一个无法立即分配给它的资源，那么当前持有的所有资源都将被释放（隐式释放）；
  - 被抢占资源被添加到进程正在等待的资源列表中；
  - 只有当进程能够恢复其原有的资源以及它所请求的新资源时，才会重新启动进程；

- **循环等待：**
  - 强制所有类型的资源进行完全排序，并要求每个进程以递增的顺序请求资源；

死锁预防策略-循环等待

- 使循环等待条件无效是最常见的。
- 只需为每个资源（即互斥锁）分配一个唯一的编号。
- 必须按顺序获得资源。
- 可以反证不存在循环等待；

假设存在资源集合R={R_1,R_2,…,R_m}，为每个资源分配一个唯一的整数，以便于比较两个资源以确定它们的先后顺序。为此，定义一个函数F:R->N，其中N是自然数的集合。例如，如果资源类型R的集合包括磁带驱动器、磁盘驱动器和打印机，那么函数F可以定义为：F(磁带驱动器) = 1；F(磁盘驱动器) = 5；F(打印机) = 12。

这样可以采用如下协议来预防死锁：每个进程只能按递增顺序申请资源。即每一个进程开始可申请任何数量的资源类型R_i的实例。之后，当且仅当$F(R_j) > F(R_i)$时，该进程才可以申请资源类型R_j的实例。例如，一个进程如果要同时使用磁带驱动器和打印机时，应首先请求磁带驱动器，然后请求打印机。换言之，要求当一个进程申请资源类型R_j时，它应先释放资源R_i （$F(R_j) \geq F(R_i)$）

死锁预防策略-循环等待

采用上述协议，循环等待就不可能成立。可通过如下反证法证明：

假设存在一个循环等待。设循环等待的进程集合为$\{P_0, P_1, \cdots, P_n\}$，其中$P_i$等待一个资源$R_i$，而$R_i$又为进程$P_{i+1}$所占有。（对索引采用取模运算，因此$P_n$等待由$P_0$所占有的资源$R_n$）。因此，由于进程$P_{i+1}$占有资源Ri而同时申请资源$R_{i+1}$，所以对于所有i，我们有$F(R_i) < F(R_{i+1})$。而这意味着$F(R_0) < F(R_1) < \cdots < F(R_n) < F(R_0)$。根据传递规则，$F(R_0) < F(R_0)$，这显然是不可能的。因此不可能有循环等待。

死锁预防策略-循环等待

如果：

F(first_mutex)=1，
F(second_mutex)=5，

那么右图中thread_two无法按错误顺序来申请锁。

设计一个完全排序或层次结构本身不能防止死锁，而是靠程序员按照顺序编写程序；

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

死锁预防策略-循环等待

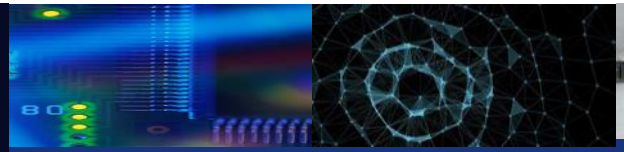
如果能够动态获得锁，那么制定一个加锁的顺序并不保证死锁预防

transaction(checking_account, savings_account, 25.0) P1

transaction(savings_account, checking_account, 50.0) P2

如果两个线程同时调用函数transaction在不同账户之间转账，有可能死锁

```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);

   acquire(lock1);
      acquire(lock2);

         withdraw(from, amount);
         deposit(to, amount);

      release(lock2);
   release(lock1);
}
```



死锁避免

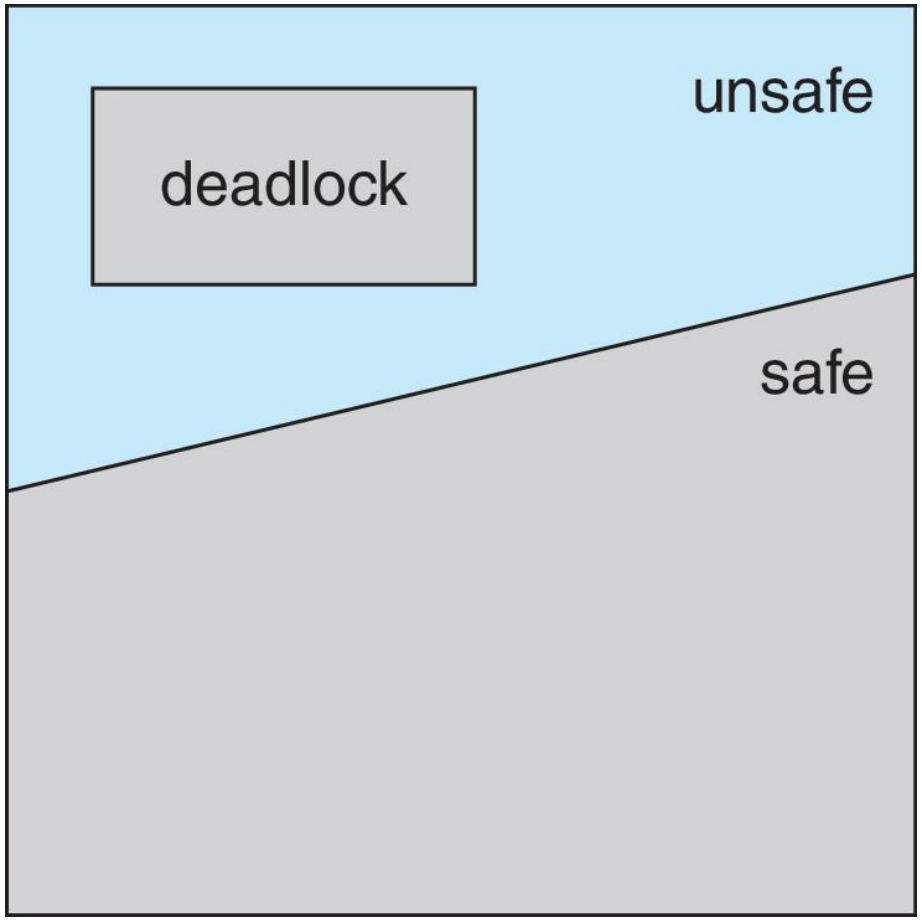
- 死锁预防的副作用是设备使用率低，系统吞吐率低；
- 要求系统有一些额外的先验信息可用；
- 最简单和最有用的模型要求每个进程声明它可能需要的每种类型的最大资源数；
- 死锁避免算法动态检查资源分配状态，以确保不存在循环等待条件；
- 资源分配状态由可用和已分配资源的数量以及进程的最大需求定义；

安全状态

- 当进程请求可用资源时，系统必须决定立即分配是否使系统处于安全状态；
- 只有存在一个安全序列，系统才处于安全状态；
- 如果系统中存在所有进程的序列<P1，P2，…，Pn>，则在当前分配状态下为安全序列是指：对于每个进程Pi，Pi仍然可以申请的资源数小于当前可用资源加上所有进程Pj（j<i）所占有的资源；
- 即：
  - 如果Pi资源需求不能立即可用，则Pi可以等待所有Pj完成，释放资源；
  - 当Pj完成时，Pi可以获得所需的资源、执行、返回分配的资源并终止；
  - 当Pi终止时，Pi+1可以获得它所需的资源，依此类推；

安全状态

安全状态不是死锁状态，相反，死锁状态是非安全状态。不是所有的非安全状态都能导致死锁状态；非安全状态可能导致死锁；



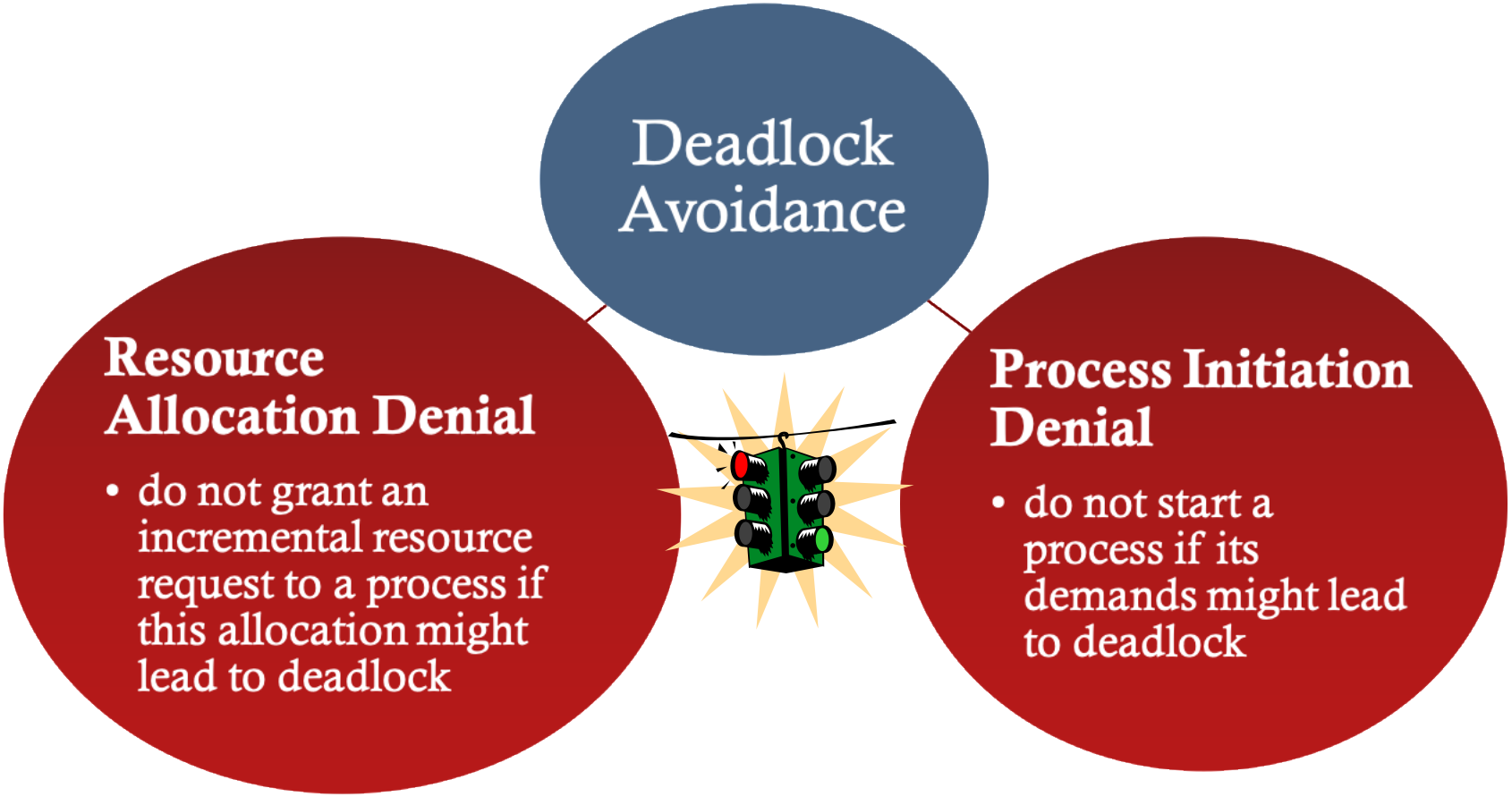
安全状态

假设系统有12台磁带驱动器和3个进程P0，P1，P2，进程P0最多要求10台，P1要求4台，P2最多要求9台。假设在时间t0时，进程P0占有5台，进程P1占有2台，进程P2占有2台。还有三台空闲，此时系统处于安全状态，因为序列<P1,P0，P2>满足安全条件。

	Maximum Needs	CurrentNeeds
P0	10	5
P1	4	2
P2	9	2

可能存在非安全状态。需确保系统始终处于安全状态，只有在分配后系统仍处于安全状态，才能允许申请。

死锁避免的两种方法


Deadlock Avoidance
Resource Allocation Denial
• do not grant an incremental resource request to a process if this allocation might lead to deadlock
Process Initiation Denial
• do not start a process if its demands might lead to deadlock

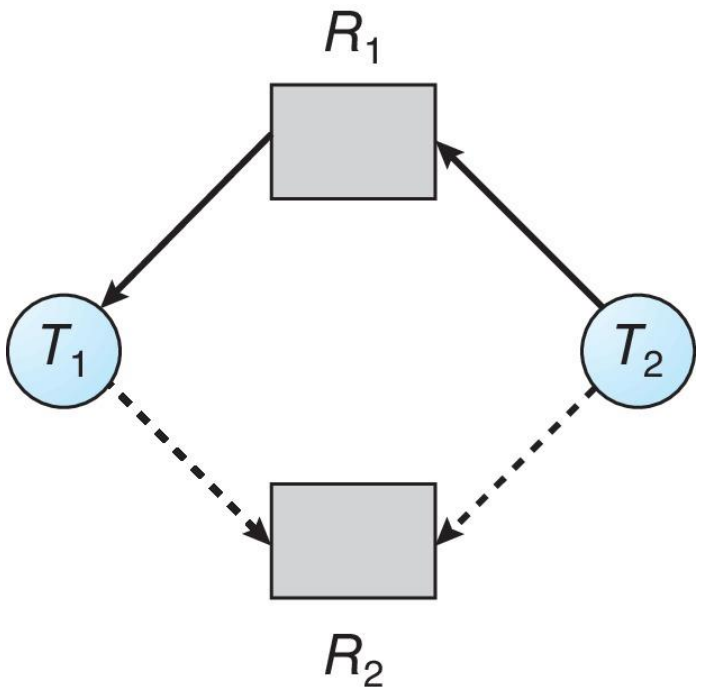
资源分配图算法

- 需求边Pi-> Rj表示进程Pi可以请求资源Rj；用虚线表示；
- 当进程请求资源时，需求边转换为请求边；
- 将资源分配给进程时，请求边转换为分配边；
- 当进程释放资源时，分配边将重新转换为需求边；
- 必须在系统中预先声明资源；
- 只有当进程Pi的所有边都为需求边时，才能允许将需求边增加到图中；

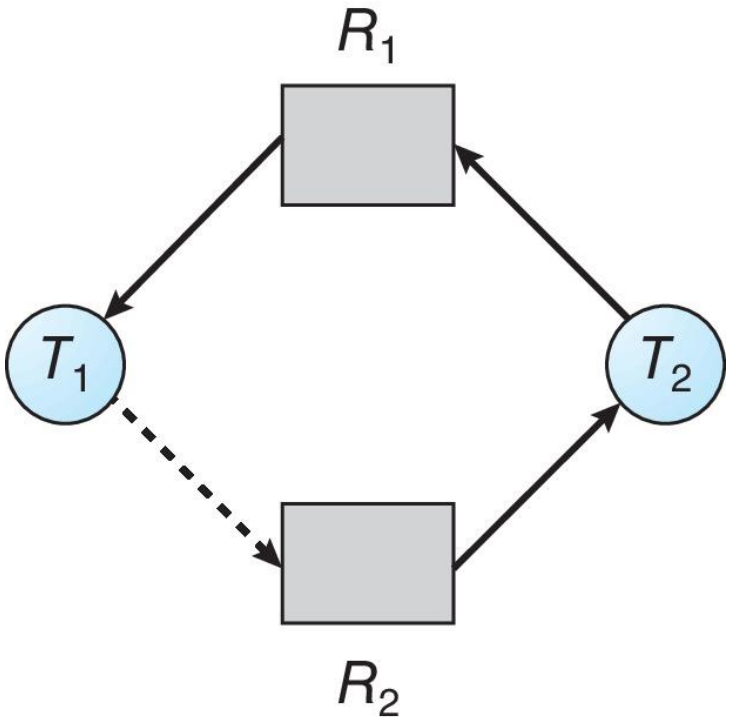
每种资源类型只有一个实例

资源分配图算法

- 假设进程Pi请求资源Rj;
- 仅当将申请边转换为分配边不会导致资源分配图中形成循环时，才能授予请求，利用环检测算法，检查安全性;



死锁避免的资源分配图



资源分配的非安全状态

银行家算法

- 每种资源类型有多个实例的资源分配系统，无法使用资源分配图；银行家算法（资源分配拒绝）；在银行中，客户申请贷款的数量是有限的，每个客户在第一次申请贷款时要声明完成该项目所需的最大资金量，在满足所有贷款要求时，客户应及时归还。银行家在客户申请的贷款数量不超过自己拥有的最大值时，都应尽量满足客户的需要。在这样的描述中，银行家就好比操作系统，资金就是资源，客户就相当于要申请资源的进程

- **每个过程都必须先验地给出最大需求资源的数量；**

- **当进程请求资源时，如果不能进入安全状态，它必须等待；**

- **当一个进程获得所有资源时，它必须在有限的时间内返回它们；**

银行家算法的数据结构

设n=进程数，m=资源类型数。

- Available：长度为m的向量，表示每种资源类型的可用实例数量。如果Available[j]=k，则有k个资源类型Rj的实例可用；
- Max:n × m矩阵，定义每个进程的最大需求。如果Max[i，j]=k，那么进程Pi最多可以请求k个资源类型Rj的实例；
- Allocation：n × m矩阵,定义每个进程现在分配的每种资源类型的实例数量。如果Allocation[i，j]=k，则Pi当前已分配了资源类型Rj的k个实例；
- Need：n × m矩阵，表示每个进程还需要的剩余资源。如果Need[i，j]=k，那么Pi还可能申请k个Rj实例来完成它的任务。注意Need[i，j]=Max[i，j]–Allocation[i，j]

银行家算法的数据结构

为了简化银行家算法的描述，下面引入一些符号。设X和Y是长度为n的向量，且定义$X \leq Y$当且仅当对所有$i=1, 2, \ldots, n$，有$X[i] \leq Y[i]$。例如，如果$X=(1, 7, 3, 2)$而$Y=(0, 3, 2, 1)$，那么$Y \leq X$。此外，如果$Y \leq X$且$Y \neq X$，那么$Y < X$。

可以将矩阵Allocation和Need的每行作为向量，并分别用$Allocation_i$和$Need_i$来表示。向量$Allocation_i$表示分配给进程P_i的资源，而向量$Need_i$表示进程为完成任务可能仍需要申请的资源

安全算法

1. **Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:**

   Work* = *Available

   ***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. **Find an *i* such that both:**

   (a) ***Finish* [*i*] = *false***

   (b) $Need_i \leq Work$

   If no such ***i*** exists, go to step 4

3. $Work = Work + Allocation_i$

   Finish*[*i*] = *true

   go to step 2

4. **If *Finish* [*i*] == *true* for all *i*, then the system is in a safe state**

算法的复杂度？ **$m \times n^2$**

资源请求算法

***Request*$_i$ = request vector for process P_i. If *Request*$_i$ [*j*] = *k* then process P_i wants *k* instances of resource type R_j**

1. If ***Request*$_i$ ≤ *Need*$_i$** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If ***Request*$_i$ ≤ *Available***, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe ⇒ the resources are allocated to P_i
- If unsafe ⇒ P_i must wait, and the old resource-allocation state is restored

银行家算法举例

- 假设有一个系统，它有5个进程P_0 到 P_4;
  3种类型资源:
  A (10 instances), B (5instances), and C (7 instances)
- 假定在时间 T_0, 系统状态如下:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

银行家算法举例

- 矩阵*Need* 的内容定义成*Max – Allocation*

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- 此时我们认为这个系统处于安全状态。因为序列 < P_1, P_3, P_4, P_2, P_0> 满足安全要求。

银行家算法举例

- 假定进程P_1再请求1个资源类型A和2个资源类型C，这样$Request_i = (1, 0, 2)$。为了确定这个请求是否可以立即允许，首先检测$Request_i \leq Available$ (即$(1,0,2) \leq (3,3,2) \Rightarrow$ true)。假定这个请求被满足，会产生如下状态

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- 序列< P_1, P_3, P_4, P_0, P_2> 满足安全要求，可以立即满足P_1的请求
- 然而当系统处于这一状态时，不能允许P_4的请求(3, 3, 0)，因为没有足够的资源可用。另外，也不能允许P_0的请求(0,2,0) 。此时虽然资源可用，但是这会导致系统处于非安全状态

确定安全状态

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(a) Initial state

上图显示了一个由四个进程和三个资源类别组成的系统状态。 资源类别 R1、R2 和 R3 分别为 9、3 和 6 个单位。 在当前状态下，已对四个进程进行分配，留下 1 个单元的 R2 和 1 个单元的 R3 可用。 这是一个安全的状态吗？ 为了回答这个问题，我们问一个中间问题：四个进程中的任何一个都可以使用可用资源运行完成吗？ 也就是说，可用资源是否可以满足任何进程的最大要求和当前分配之间的差异？ 就前面介绍的矩阵和向量来说，要满足的条件对于进程是$C_{ij} - A_{ij} \leq V_{ij}$

确定安全状态

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) P2 runs to completion

显然，这对 P1 来说是不可能的，它只有 1 个 R1 单元，还需要 2 个 R1 单元、2 个 R2 单元和 2 个 R3 单元。 但是，通过将一个单元的 R3 分配给进程 P2，P2 已分配了其所需的最大资源并可以运行至完成。 假设这已经完成。 当 P2 完成时，它的资源可以返回到可用资源池中。 结果状态如图所示。 现在我们可以再次询问是否可以完成任何剩余过程。 在这种情况下，可以完成剩余的每个过程。

确定安全状态

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

(c) P1 runs to completion

假设我们选择P1，分配需要的资源，完成P1，将P1的所有资源归还到可用池中。此时处于上图 所示的状态。

确定安全状态

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

确定非安全状态

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

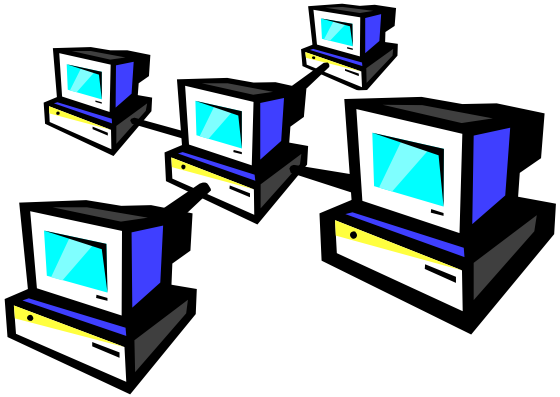
R1	R2	R3
0	1	1

Available vector **V**

(b) P1 requests one unit each of R1 and R3

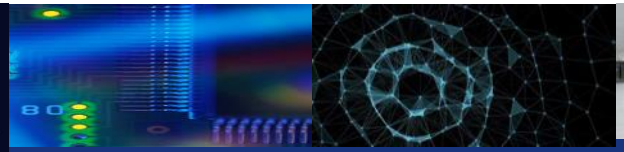
死锁避免的优势

- 无须死锁预防中的抢占和回滚进程；
- 与死锁预防相比限制较少；



死锁避免的限制

- 必须事先声明每一个进程请求的最大资源；
- 所讨论的进程必须是无关的，即他们的执行顺序必须没有任何同步要求的限制；
- 分配的资源数量必须是固定的；
- 在占有资源时，进程不能退出；



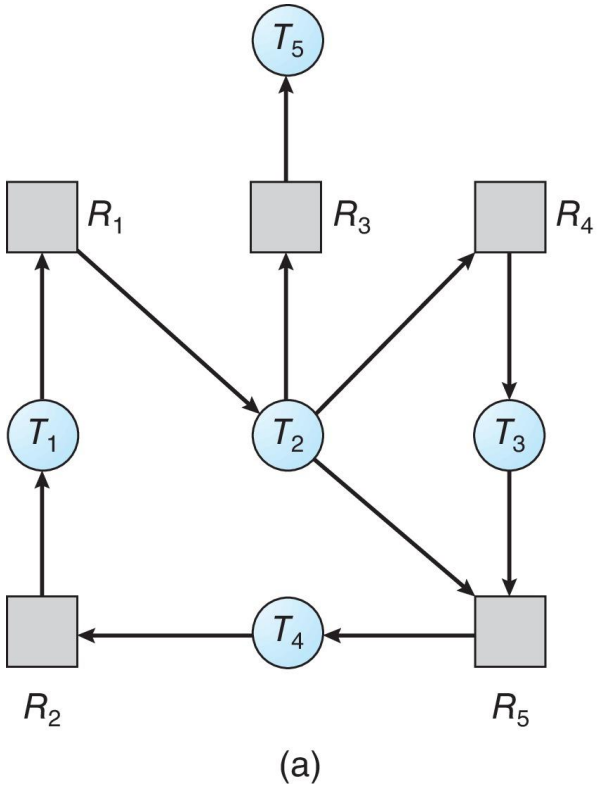
死锁检测

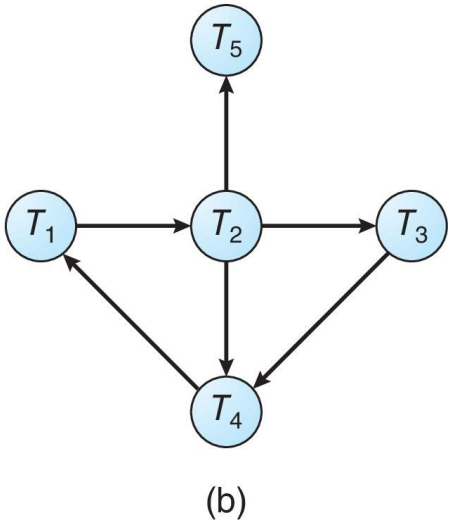
- 允许系统进入死锁状态
- 检测算法
- 回收计划

每个资源类型有单个实例

- 维护等待图
  - 节点是进程
  - Pi-> Pj如果Pi正在等待Pj；
- 定期调用在图中搜索循环的算法。如果存在循环，则存在死锁；
- 在图中检测循环的算法需要n^2次操作，其中n是图中的顶点数

等待图





资源分配图　　对应等待图

每个资源类型有多个实例

类似银行家算法的数据结构：

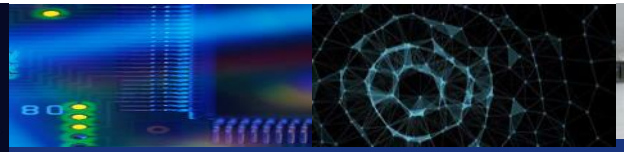
- 可用：长度为 m 的向量表示每种类型的可用资源数量；
- 分配：n x m矩阵定义了当前分配给每个进程的每种类型的资源数量；
- 请求：n x m矩阵表示每个进程的当前请求。如果请求[i][j]=k，则进程Pi正在请求更多资源类型Rj的k个实例；

每个资源类型有多个实例

1. **Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:**
   a) ***Work* = *Available***
   b) For ***i* = 1,2, …, *n***, if ***Allocation*$_i$ ≠ 0**, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***

2. **Find an index *i* such that both:**
   a) ***Finish*[*i*] == *false***
   b) ***Request*$_i$ ≤ *Work***

   If no such ***i*** exists, go to step 4



每个资源类型有多个实例

3. *Work* = *Work* + *Allocation*$_i$
   Finish[*i*] = *true*
   go to step 2

4. If *Finish[i]* == *false*, for some *i*, $1 \le i \le n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state

死锁检测样例

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish[i] = true* for all *i*

死锁检测样例

- P_2 requests an additional instance of type C

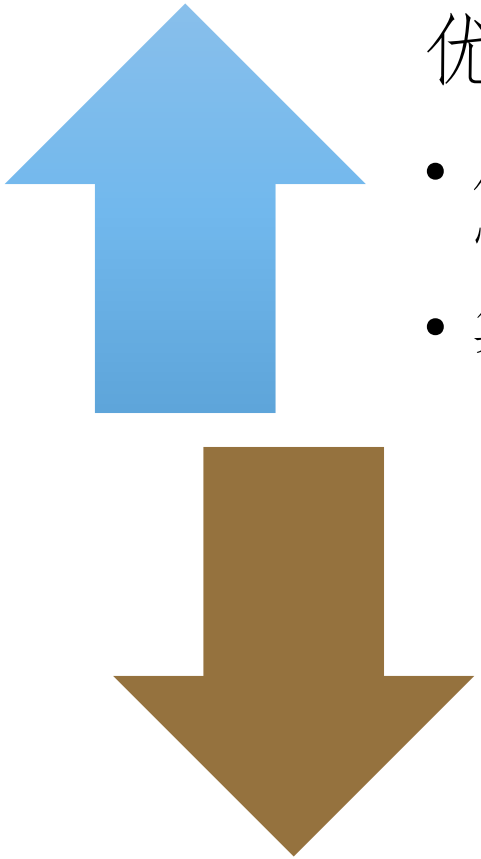
Request

	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- **State of system?**
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

死锁检测算法

- 死锁检测可以频繁地在每个资源请求发生时进行，也可以进行得少一些，具体取决于发生死锁的可能性。



优点：

- 尽早地检测死锁情况；
- 算法相对简单；

缺点

- 频繁的检测会耗费相当多的处理器时间；

死锁检测算法的使用

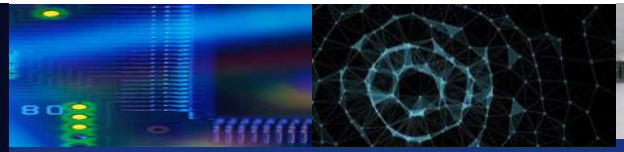
- 何时以及多久调用一次取决于：
  - 死锁可能发生的频率有多高？
  - 需要回滚多少个进程？

- 如果任意调用检测算法，资源图中可能会有许多循环，因此我们无法判断是哪些死锁进程“导致”了死锁；

- 死锁检测产生大量的代价，可以周期性检测死锁，或者CPU的利用率低于某个阈值；

死锁恢复策略

- 取消所有的死锁进程，这是操作系统最常用的方法；
- 把每个死锁进程回滚到前面定义的某些检查点，并重新启动所有进程，要求在系统中构建回滚和重启机制；
- 连续取消死锁进程直到不再存在死锁，所取消的进程的顺序应基于某种最小代价原则；
- 连续抢占资源直到不再存在死锁；需要使用一种基于代价的选择方法，且需要在抢占后重新调用检测算法。



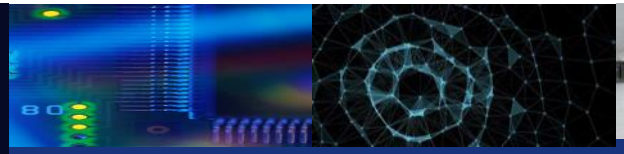
恢复策略

对于前面的（3）和（4）方法，选择原则如下：

- 目前为止消耗的处理器时间最少；
- 目前为止产生的输出最少；
- 预计剩下的时间最长；
- 目前为止分配的资源总量最少；
- 优先级最低；

资源抢占

- **Selecting a victim – minimize cost**
- **Rollback – return to some safe state, restart process for that state**
- **Starvation – same process may always be picked as victim, include number of rollback in cost factor**



Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

Table 6.1

Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLO80]

哲学家就餐问题

- No two philosophers can use the same fork at the same time (mutual exclusion)
- No philosopher must starve to death (avoid deadlock and starvation)

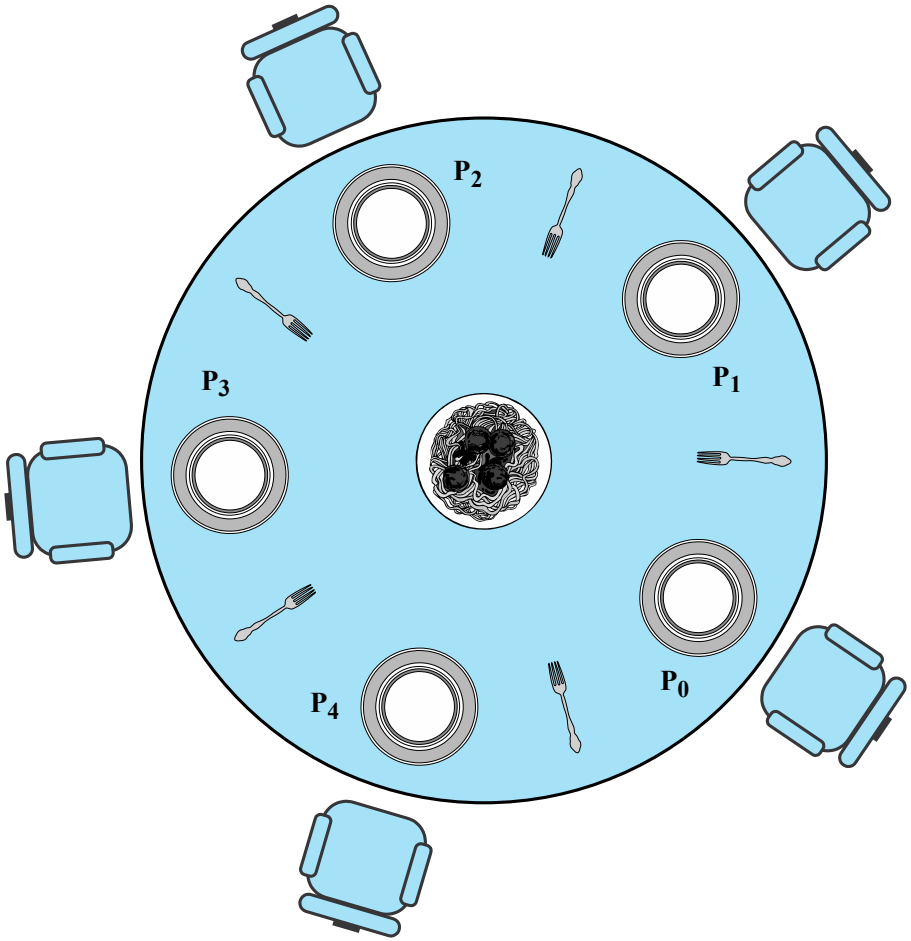


Figure 6.11 Dining Arrangement for Philosophers

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher
(2),
          philosopher (3), philosopher (4));
     }
```

Figure 6.12　A First Solution to the Dining Philosophers Problem

当所有的哲学家都饿了而且都拿起了左侧的筷子或叉子，将会产生死锁和饥饿

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
      think();
      wait (room);
      wait (fork[i]);
      wait (fork [(i+1) mod 5]);
      eat();
      signal (fork [(i+1) mod 5]);
      signal (fork[i]);
      signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```

可以考虑增加一名服务员，一次只允许四位哲学家进入餐厅。 最多有四个坐着的哲学家，至少有一个哲学家可以使用两个叉子

Figure 6.13 A Second Solution to the Dining Philosophers Problem

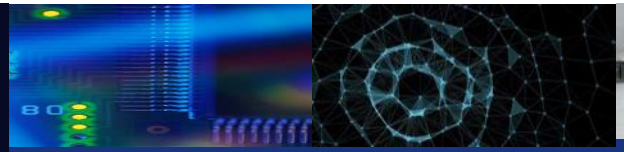
```
monitor dining_controller;
cond ForkReady[5];          /* condition variable for synchronization */
boolean fork[5] = {true};       /* availability status of each fork */

void get_forks(int pid)          /* pid is the philosopher id number */
{
   int left = pid;
   int right = (++pid) % 5;
   /*grant the left fork*/
   if (!fork[left])
      cwait(ForkReady[left]);         /* queue on condition variable */
   fork[left] = false;
   /*grant the right fork*/
   if (!fork[right])
      cwait(ForkReady[right]);        /* queue on condition variable */
   fork[right] = false:
}
void release_forks(int pid)
{
   int left = pid;
   int right = (++pid) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left])      /*no one is waiting for this fork */
      fork[left] = true;
   else                       /* awaken a process waiting on this fork */
      csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])     /*no one is waiting for this fork */
      fork[right] = true;
   else                       /* awaken a process waiting on this fork */
      csignal(ForkReady[right]);
}
```

```
void philosopher[k=0 to 4]           /* the five philosopher clients */
{
   while (true) {
      <think>;
      get_forks(k);          /* client requests two forks via monitor */
      <eat spaghetti>;
      release_forks(k);      /* client releases forks via the monitor */
   }
}
```

Figure 6.14

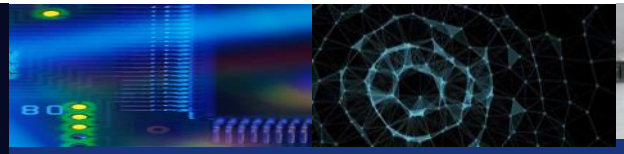
A Solution to the Dining Philosophers Problem Using a Monitor

与信号量解决方案不同，管程的解决方案不会发生死锁，因为一次只有一个进程可能处于管程中。 例如，第一个进入管程的哲学家进程被保证在它拿起左叉子之后，他可以在下一个哲学家拿起它的左叉子之前拿起右边的叉子，这里右边的叉子即为另外可能哲学家的左边的叉子



本章小结

- 什么是死锁？
- 死锁特征：互斥、占有等待、非抢占、循环等待
- 资源分配图表征死锁
- 死锁处理方法：死锁预防(prevention)—资源类型排序、死锁避免(avoidance)—安全状态、资源分配图算法、银行家算法
- 死锁检测：等待图
- 死锁恢复：牺牲进程、回滚及饥饿



谢谢