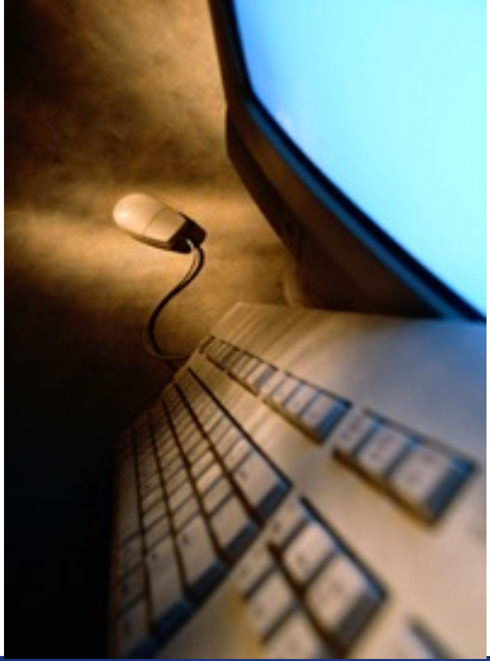
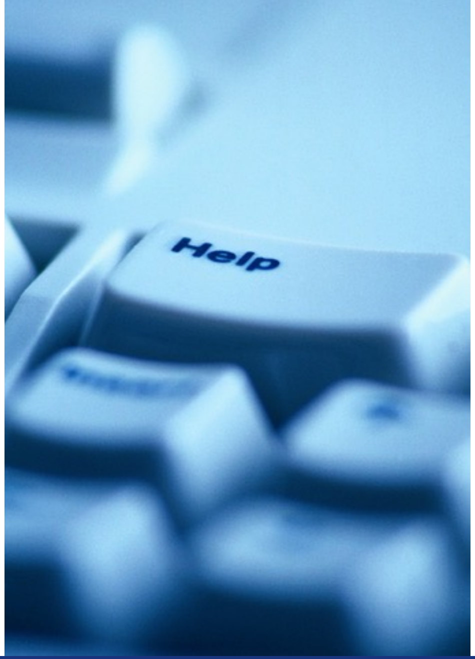



操作系统原理
Operating Systems Principles

陈鹏飞
计算机学院

第八讲 —并发: 死锁、饥饿

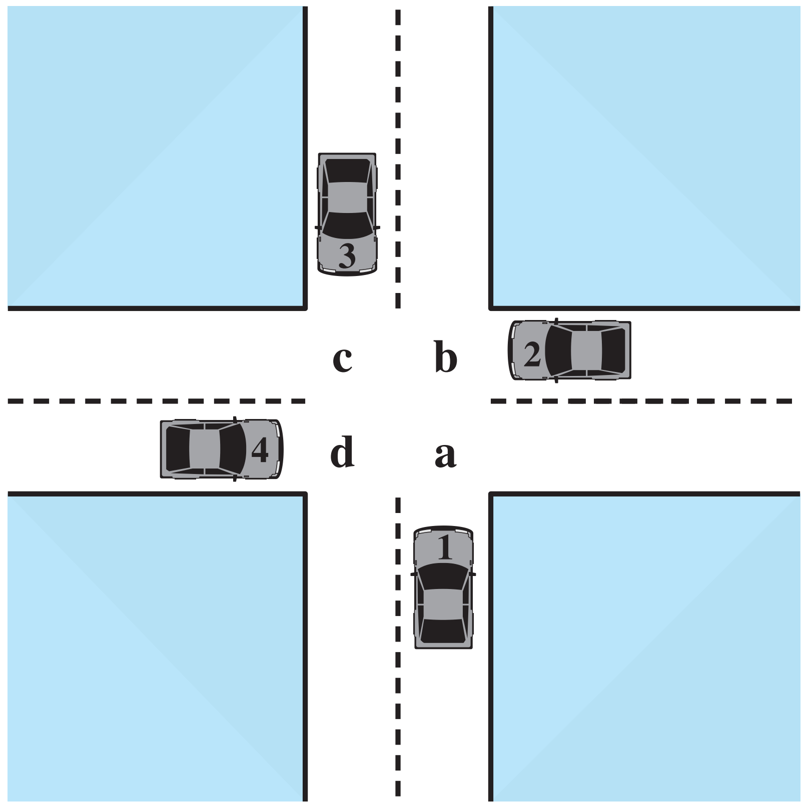
目标

- 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；

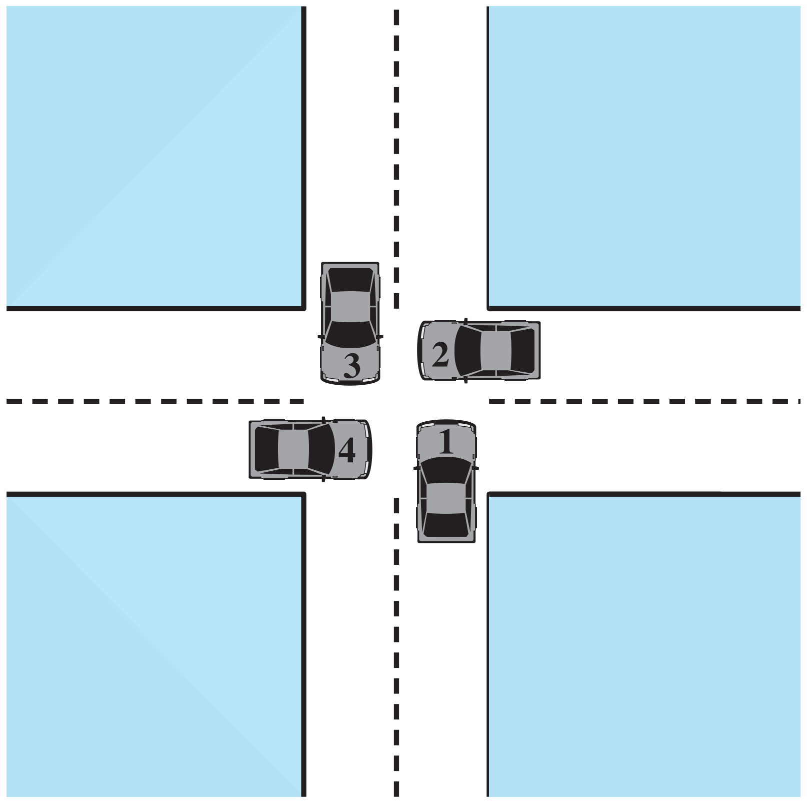
死锁

- 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)

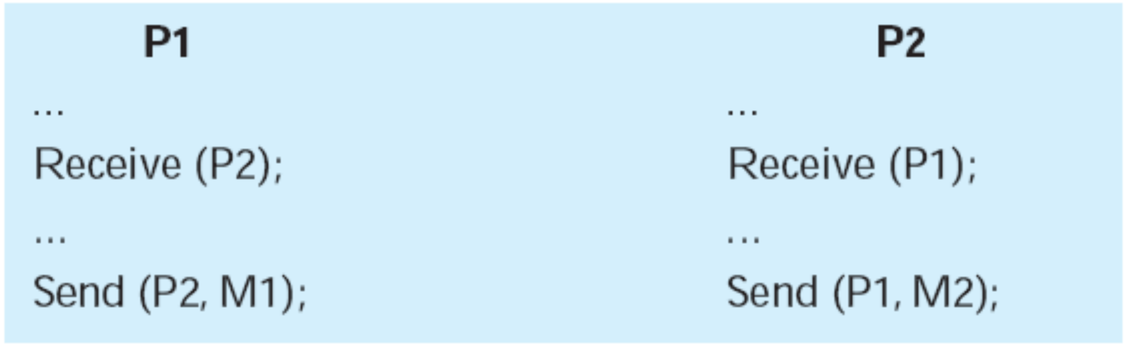
Process Q

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；

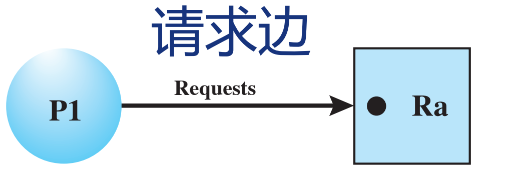
死锁的必要条件

死锁的存在性

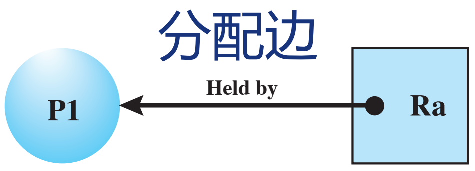
互斥	占有等待	非抢占	循环等待
• 一次只有一个进程使用一个资源，其他进程不能访问分配给其他进程的资源；	• 当一个进程等待其他进程时，继续占有已分配的资源；	• 不能强行抢占进程已占有的资源；	• 存在一个闭合的进程链，每个进程至少占有此链中下一个进程所需的一个资源；

死锁的可能性

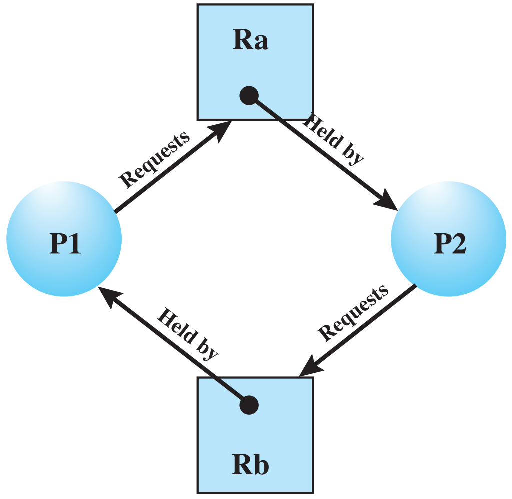
资源分配图



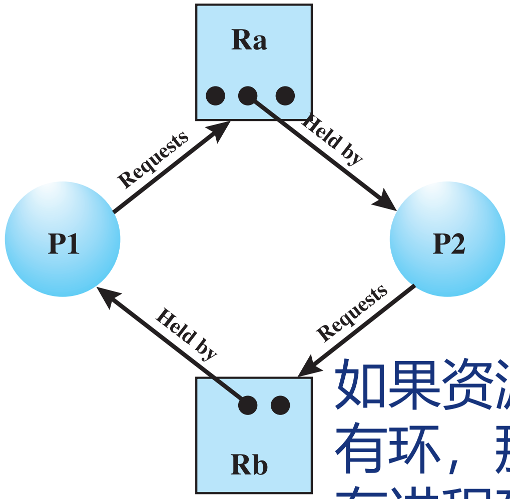
(a) Resouce is requested



(b) Resource is held



(c) Circular wait



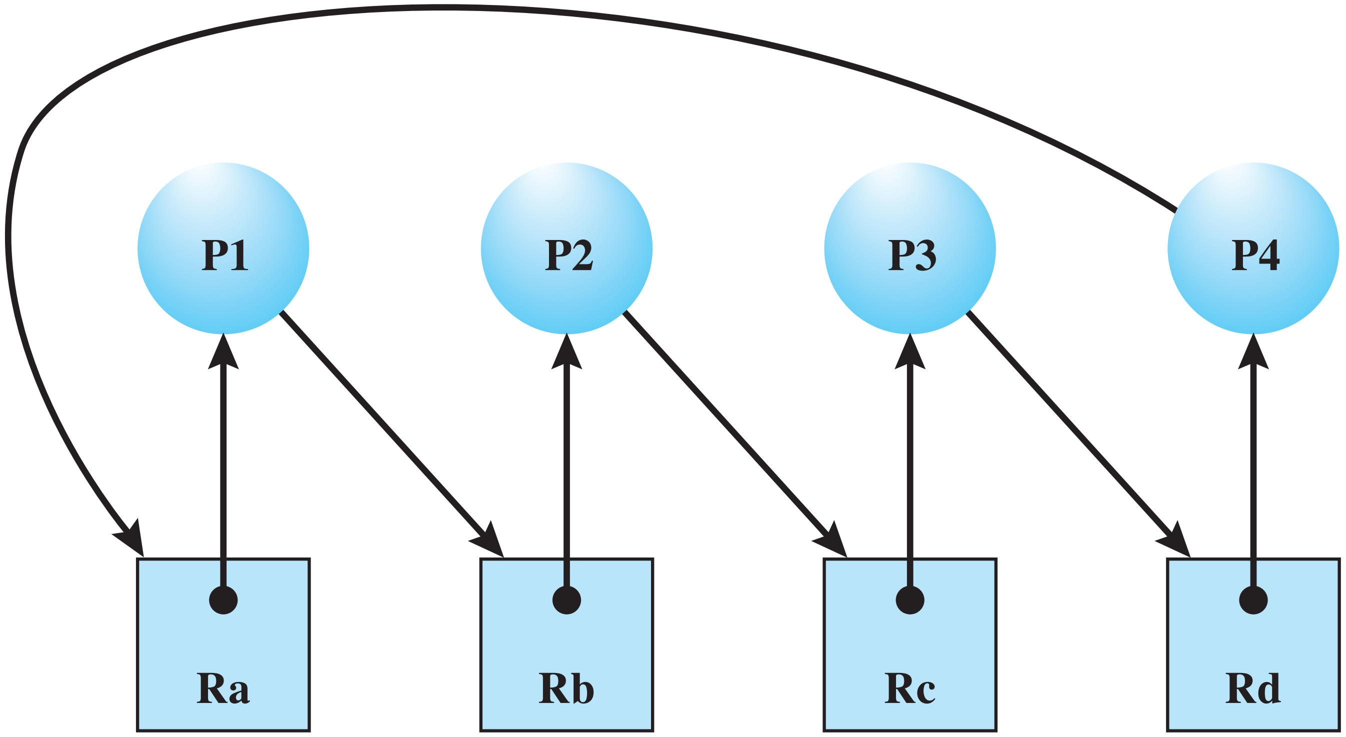
(d) No deadlock

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Holt引入的资源分配图

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
  - 死锁避免

- 允许系统进入死锁状态，然后检测和恢复；

- 忽略该问题，并假装系统中从未发生死锁；
  - 绝大多数操作系统所采用的的方法；

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；**

- **持有且等待 - 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。
  - 资源利用率低，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；

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

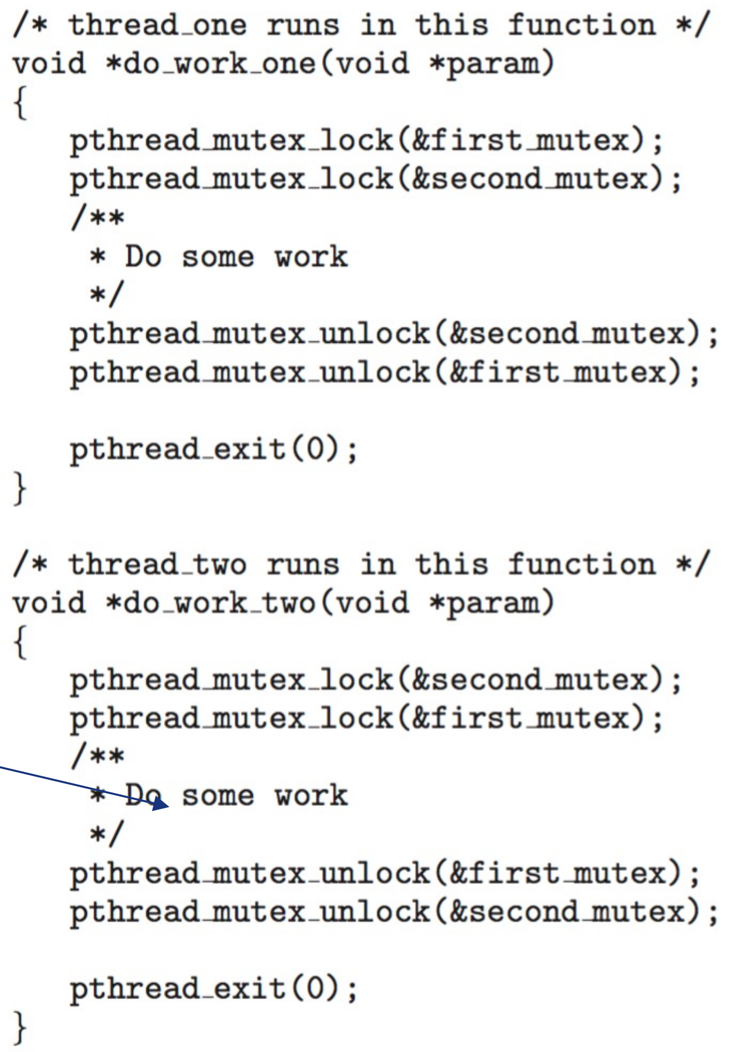
- 使循环等待条件无效是最常见的。
- 只需为每个资源（即互斥锁）分配一个唯一的编号。
- 必须按顺序获得资源。
- 可以反证不存在循环等待；

If these two protocols are used, then the circular-wait condition cannot hold. We can demonstrate this fact by assuming that a circular wait exists (proof by contradiction). Let the set of threads involved in the circular wait be $\{T_0, T_1, ..., T_n\}$, where T_i is waiting for a resource R_i, which is held by thread T_{i+1}. (Modulo arithmetic is used on the indexes, so that T_n is waiting for a resource R_n held by T_0.) Then, since thread T_{i+1} is holding resource R_i while requesting resource R_{i+1}, we must have $F(R_i) < F(R_{i+1})$ for all i. But this condition means that $F(R_0) < F(R_1) < ... < F(R_n) < F(R_0)$. By transitivity, $F(R_0) < F(R_0)$, which is impossible. Therefore, there can be no circular wait.

死锁预防策略-循环等待

如果：first_mutex=1, second_mutex=5 code for thread_two无法按如下方式写入：

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
```

```
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);
}
```

Figure 8.7 Deadlock example with lock ordering.

死锁避免

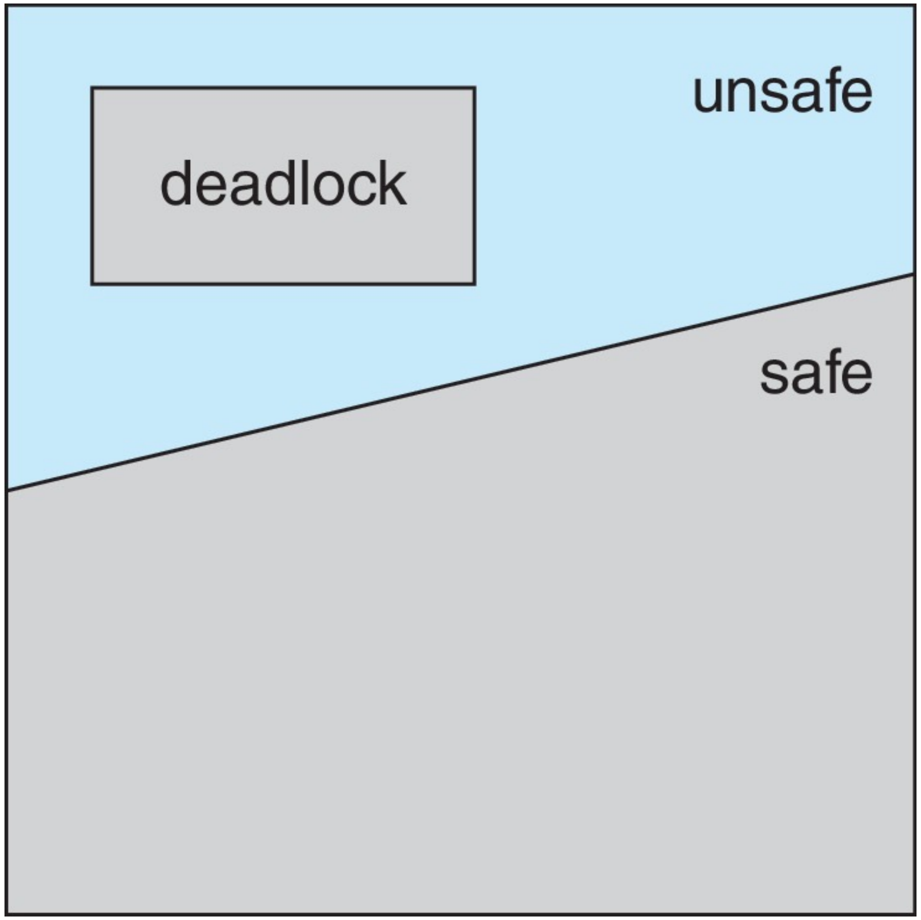
- 死锁预防的副作用是设备使用率低，系统吞吐率低；
- 要求系统有一些额外的先验信息可用例如：如何申请资源；
- 最简单和最有用的模型要求每个进程声明它可能需要的每种类型的最大资源数；
- 死锁避免算法动态检查资源分配状态，以确保不存在循环等待条件；
- 资源分配状态由可用和已分配资源的数量以及进程的最大需求定义；

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Pi，Pi请求的资源可以通过当前可用资源+所有Pj持有的资源来满足，j<i；
- 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

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。

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

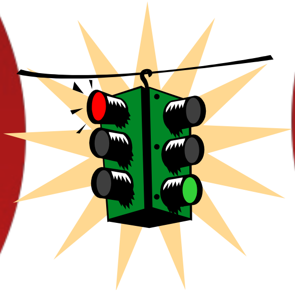
可能存在非安全状态。需确保系统始终处于安全状态，只有在分配后系统仍处于安全状态，才能允许申请。

死锁避免的两种方法

Deadlock Avoidance

Resource Allocation Denial

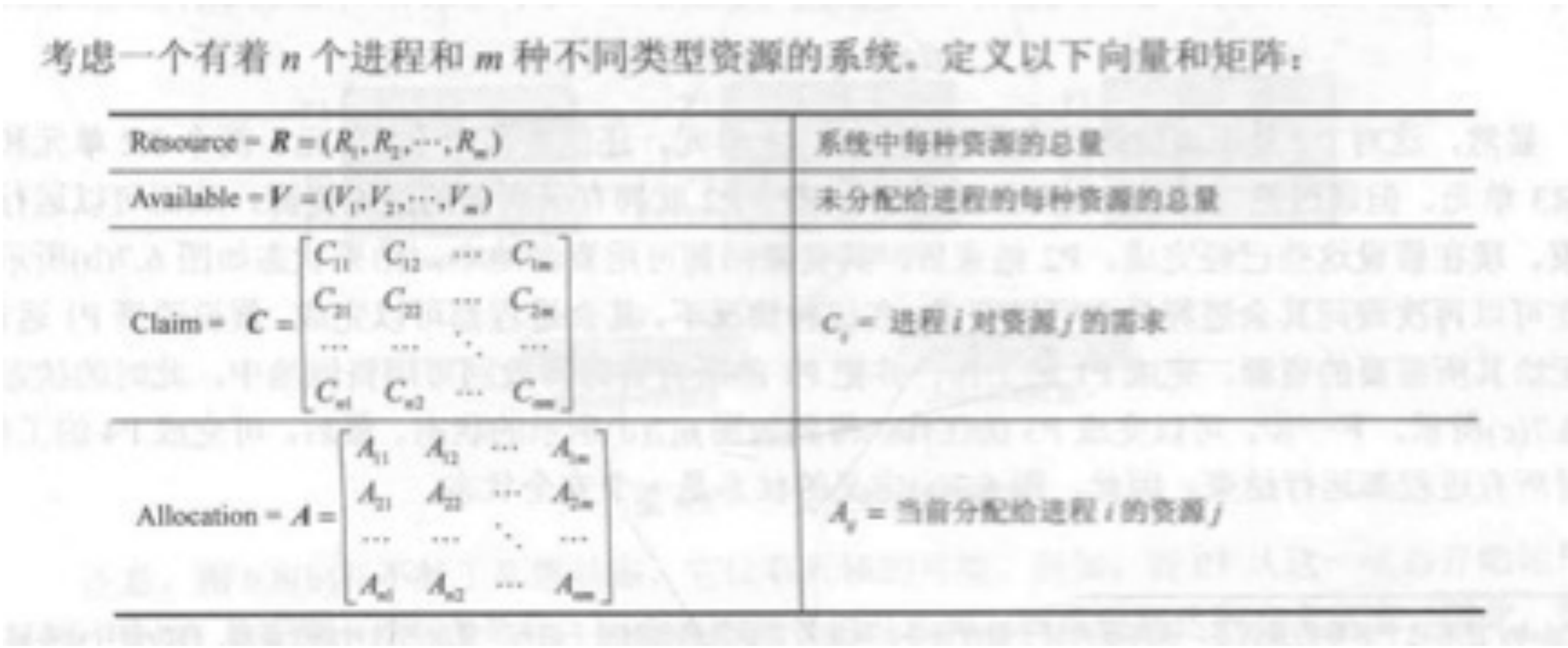
- do not grant an incremental resource request to a process if this allocation might lead to deadlock



Process Initiation Denial

- do not start a process if its demands might lead to deadlock

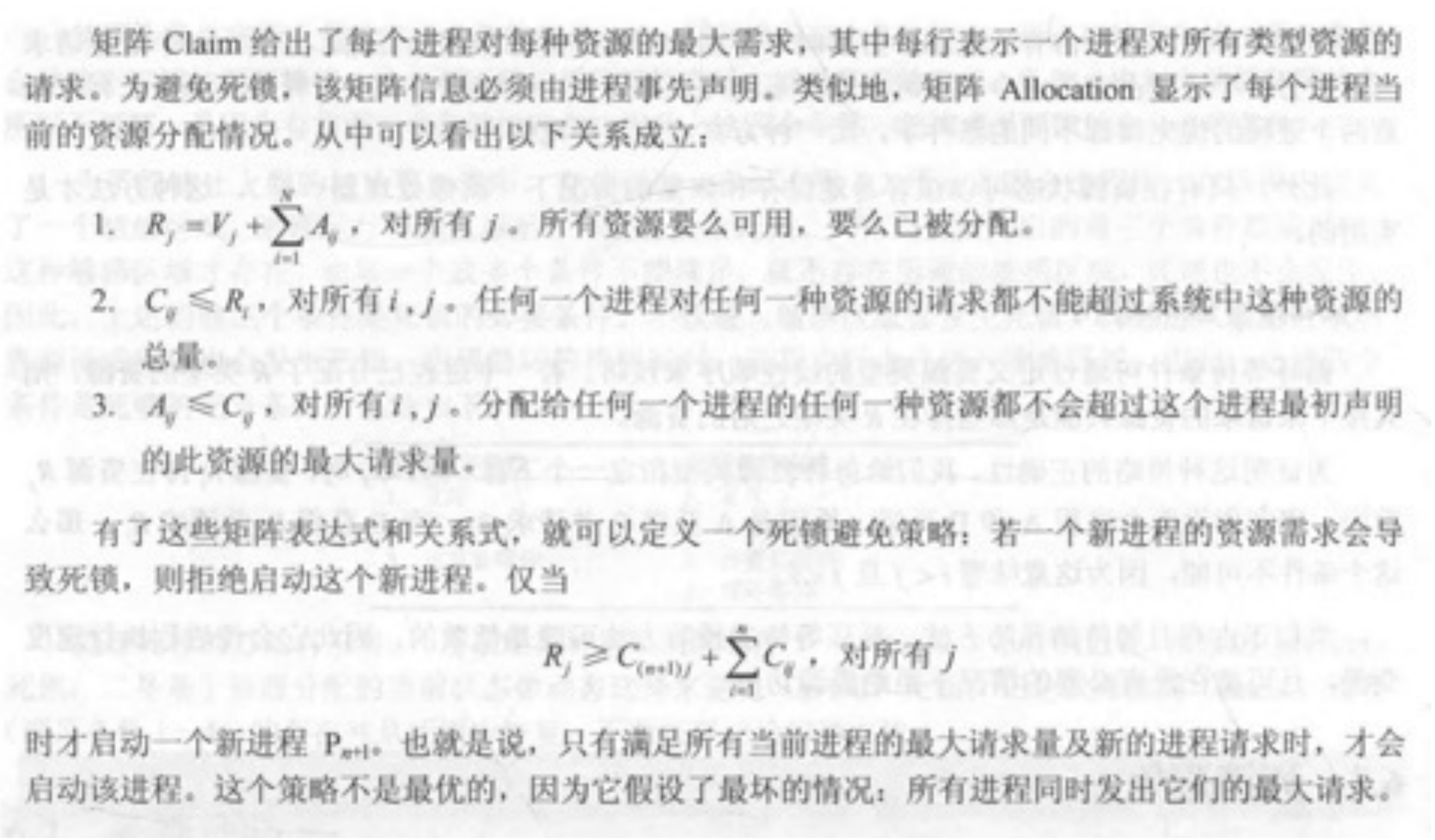
进程启动拒绝



考虑一个有着 n 个进程和 m 种不同类型资源的系统。定义以下向量和矩阵：

Resource = $\boldsymbol{R} = (R_1, R_2, \cdots, R_m)$	系统中每种资源的总量
Available = $\boldsymbol{V} = (V_1, V_2, \cdots, V_m)$	未分配给进程的每种资源的总量
Claim = $\boldsymbol{C} = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \cdots & \cdots & \ddots & \cdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{bmatrix}$	C_{ij} = 进程 i 对资源 j 的需求
Allocation = $\boldsymbol{A} = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \cdots & \cdots & \ddots & \cdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{bmatrix}$	A_{ij} = 当前分配给进程 i 的资源 j

进程启动拒绝



矩阵 Claim 给出了每个进程对每种资源的最大需求，其中每行表示一个进程对所有类型资源的请求。为避免死锁，该矩阵信息必须由进程事先声明。类似地，矩阵 Allocation 显示了每个进程当前的资源分配情况。从中可以看出以下关系成立：

1. $R_j = V_j + \sum_{i=1}^{N} A_{ij}$，对所有 j。所有资源要么可用，要么已被分配。
2. $C_{ij} \leqslant R_i$，对所有 i, j。任何一个进程对任何一种资源的请求都不能超过系统中这种资源的总量。
3. $A_{ij} \leqslant C_{ij}$，对所有 i, j。分配给任何一个进程的任何一种资源都不会超过这个进程最初声明的此资源的最大请求量。

有了这些矩阵表达式和关系式，就可以定义一个死锁避免策略：若一个新进程的资源需求会导致死锁，则拒绝启动这个新进程。仅当

$$R_j \geqslant C_{(n+1)j} + \sum_{i=1}^{n} C_{ij}，对所有 j$$

时才启动一个新进程 P_{n+1}。也就是说，只有满足所有当前进程的最大请求量及新的进程请求时，才会启动该进程。这个策略不是最优的，因为它假设了最坏的情况：所有进程同时发出它们的最大请求。

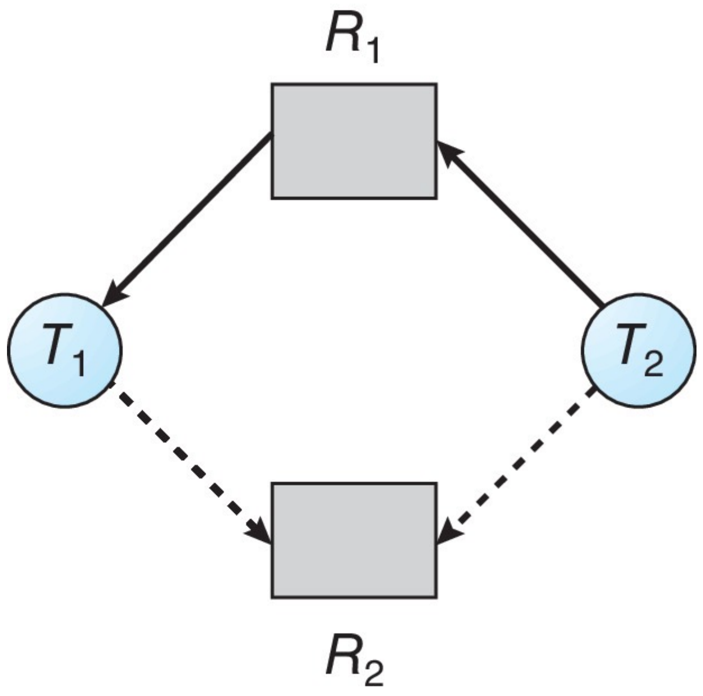
资源分配图算法

- 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;
- 当流程释放资源时，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（预先声明）；

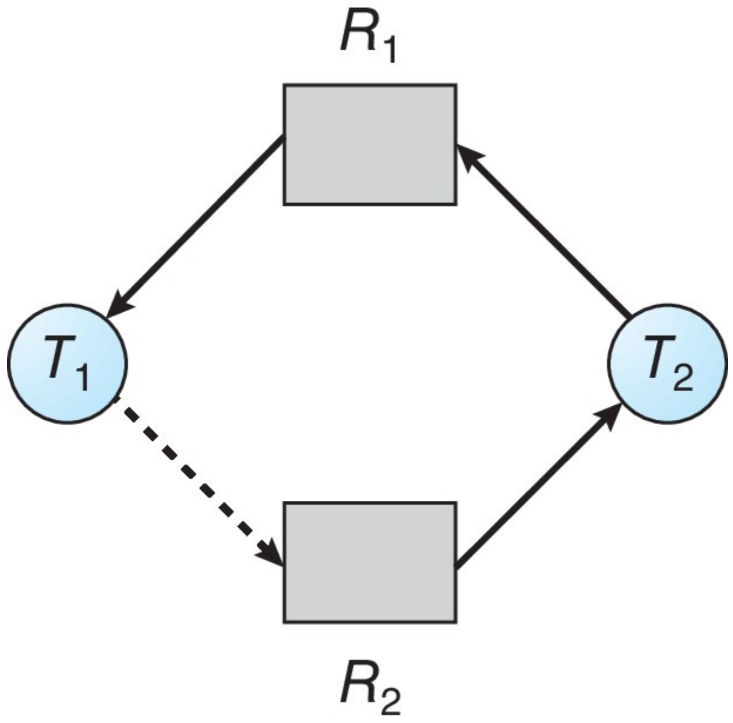
每种资源类型只有一个实例

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检测死锁；
- 银行家算法（**资源分配拒绝**）；
- **每个进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。如果Available[j]=k，则有k个资源类型Rj的实例可用；
- Max:n x 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当前分配给Rj的k个实例；
- Need：n x 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可能还需要k个Rj实例来完成它的任务 **Need[i，j]=Max[i，j] - Allocation[i，j]**

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) $\boldsymbol{Need_i \leq Work}$

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

3. $\boldsymbol{Work = Work + Allocation_i}$

   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？

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 $\boldsymbol{Request_i \le Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\boldsymbol{Request_i \le Available}$, go to step 3. Otherwise $\boldsymbol{P_i}$ must wait, since resources are not available
3. Pretend to allocate requested resources to $\boldsymbol{P_i}$ by modifying the state as follows:

$$\boldsymbol{Available = Available - Request_i;}$$
$$\boldsymbol{Allocation_i = Allocation_i + Request_i;}$$
$$\boldsymbol{Need_i = Need_i - Request_i;}$$

- If safe $\Rightarrow$ the resources are allocated to $\boldsymbol{P_i}$
- If unsafe $\Rightarrow$ $\boldsymbol{P_i}$ must wait, and the old resource-allocation state is restored

银行家算法举例

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), *B* (5instances), and *C* (7 instances)

- Snapshot at time T_0:

	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

- **The content of the matrix *Need* is defined to be *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**

- **The system is in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfies safety criteria**

银行家算法举例

- Check that P1 Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

	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	

- Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

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

Figure 6.7 Determination of a Safe State

确定安全状态

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

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

Resource vector **R**

R1	R2	R3
9	3	6

Available vector **V**

R1	R2	R3
6	2	3

(b) P2 runs to completion

Figure 6.7 Determination of a Safe State

确定安全状态

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

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

Resource vector **R**

R1	R2	R3
9	3	6

Available vector **V**

R1	R2	R3
7	2	3

(c) P1 runs to completion

Figure 6.7 Determination of a Safe State

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**

Figure 6.7 Determination of a Safe State

确定非安全状态

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

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

Resource vector **R**

R1	R2	R3
9	3	6

Available vector **V**

R1	R2	R3
1	1	2

(a) Initial state

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

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

Resource vector **R**

R1	R2	R3
9	3	6

Available vector **V**

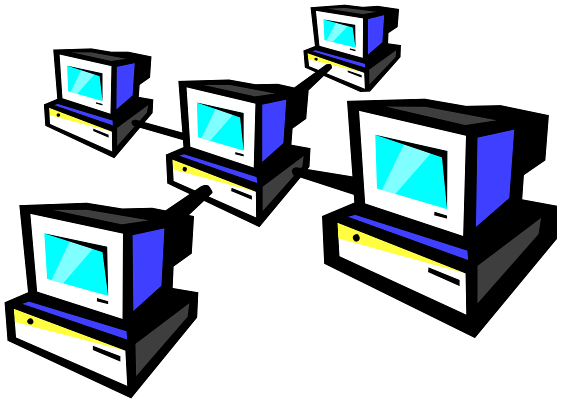
R1	R2	R3
0	1	1

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

Figure 6.8 Determination of an Unsafe State

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；

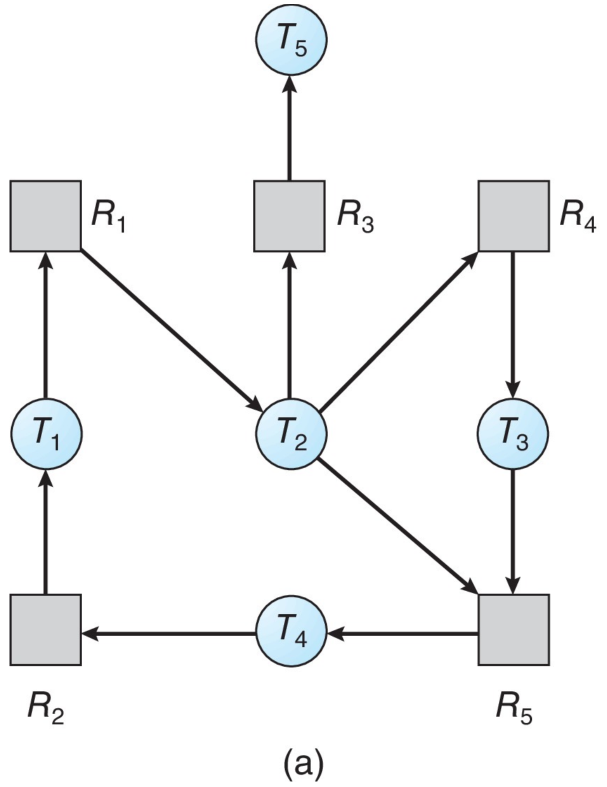
死锁检测

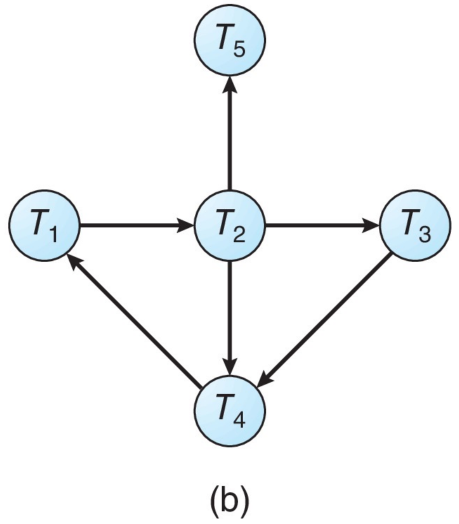
- 允许系统进入死锁状态
- 检测算法
- 回收计划

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：nxm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：nxm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 ≤ *i* ≤ *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)

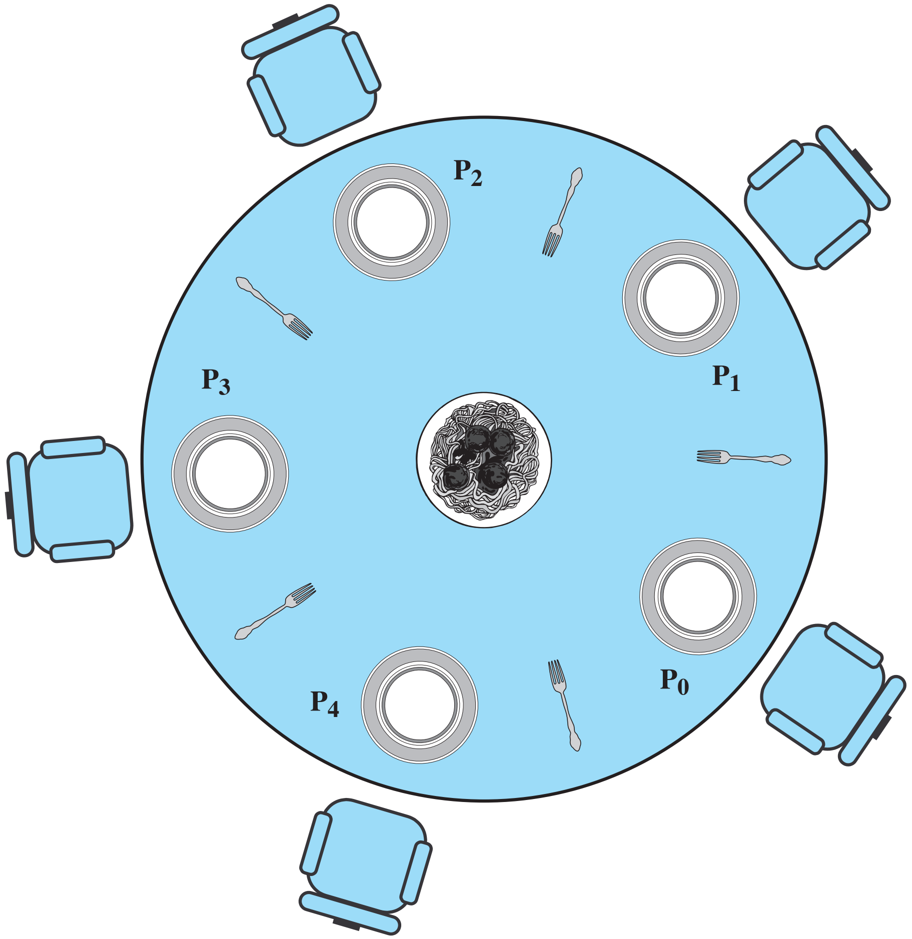


Figure 6.11 Dining Arrangement for Philosophers

```
/* 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

```
/* 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));
}
```

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

谢谢