第6章 死锁

- 本章内容
- 1. 死锁的定义、产生的原因、必要条件
- 2. 解决死锁的典型方法：死锁预防、死锁避免、死锁检测、死锁解除
- 3. 以哲学家进餐问题为例，了解导致死锁的原因，以及怎样解决死锁

系统模型

- 系统有m类资源$R_1, R_2, \ldots, R_m$
  - CPU周期，内存空间，I/O设备
- 每一类资源R_i有W_i个实例
- 进程按如下方式利用资源
  - 申请 request
  - 使用 use
  - 释放 release

死锁示例

- 线程A和线程B共享两把全局互斥锁A和B

```
void proc_A(void){
   lock(A);
   // t0时刻
   lock(B);
   // 临界区
   unlock(B);
   unlock(A);
}
```

```
void proc_B(void){
   lock(B);
   // t0时刻
   lock(A);
   // 临界区
   unlock(A);
   unlock(B);
}
```

- 假设在t_0时刻，线程A获得了锁A，将要尝试获得锁B。线程B获得了锁B，正等待获取锁A
- 两个线程都无法获得对方持有的锁，无法进入临界区，此时的状态我们称之为死锁

死锁定义

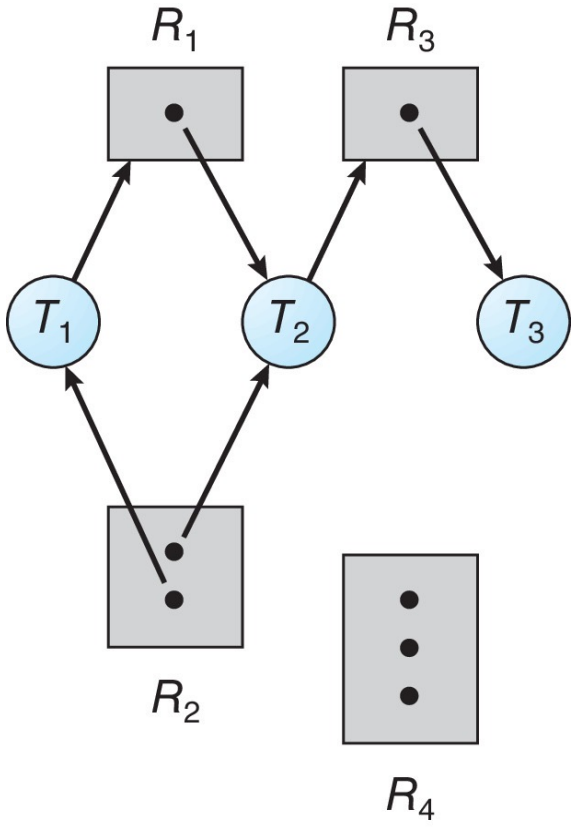
- 当有多个（两个及以上）线程为有限资源竞争时，有的线程就会因为在某一时刻没有可用的空闲资源而陷入等待。死锁(deadlock)就是指这一组中的每个线程都在等待组内其他线程释放资源从而造成的无限等待。若无外力作用，这些进程将永远都不能向前推进。
- 注意：
  - 死锁是因资源竞争造成的僵局
  - 通常死锁至少涉及两个进程
  - 死锁与部分进程及资源相关

资源分配图

- 资源分配图由一个节点集合V和一个边集合E组成
  - V被分为两个部分
    - 系统中全部进程集合P={P1, P2, …, Pn}
    - 系统中全部资源集合R= {R1, R2, …, Rm}
  - 请求边：有向边$P_i \rightarrow R_j$;
  - 分配边：有向边 $R_j \rightarrow P_i$
- 在图中，用圆形表示进程P_i，用方框表示资源类型R_j，在方框中用圆点表示实例

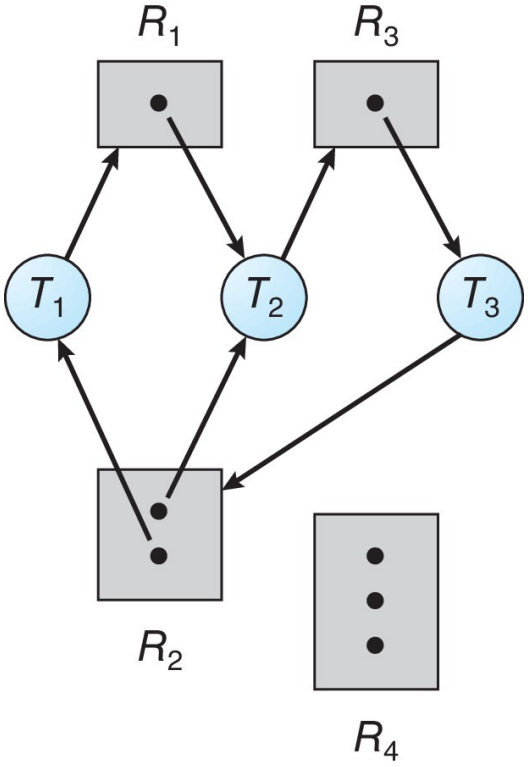
资源分配图示例

- R_1有一个实例
- R_2 有两个实例
- R_3 有一个实例
- R_4 有三个实例
- T_1 占有R_2 的一个实例且等待 R_1 的一个实例
- T_2 占有R_1 的一个实例, R_2 的一个实例, 且等待 R_3 的一个实例
- T_3 占有R_3 的一个实例

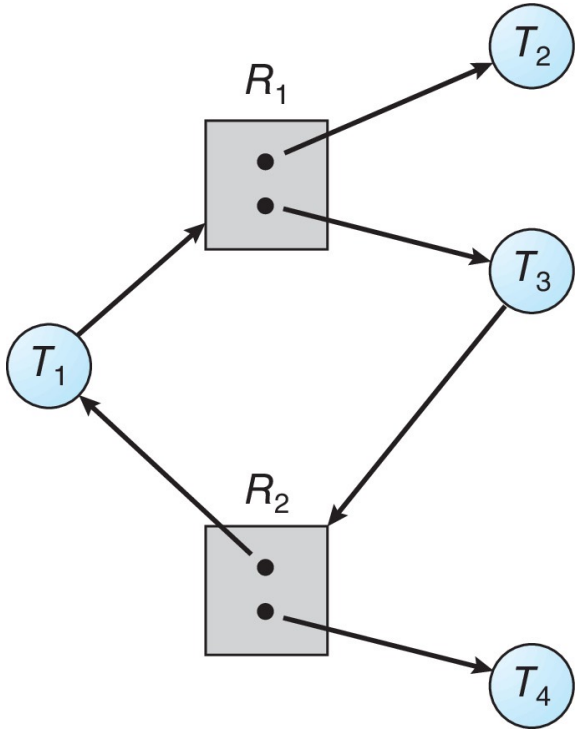


带环的资源分配图

- 可以证明：如果资源分配图没有环，那么系统就没有进程死锁。如果分配图有环，每类资源仅有一个实例，则存在死锁，若每类资源有多个实例，则可能存在死锁。



存在死锁的资源分配图



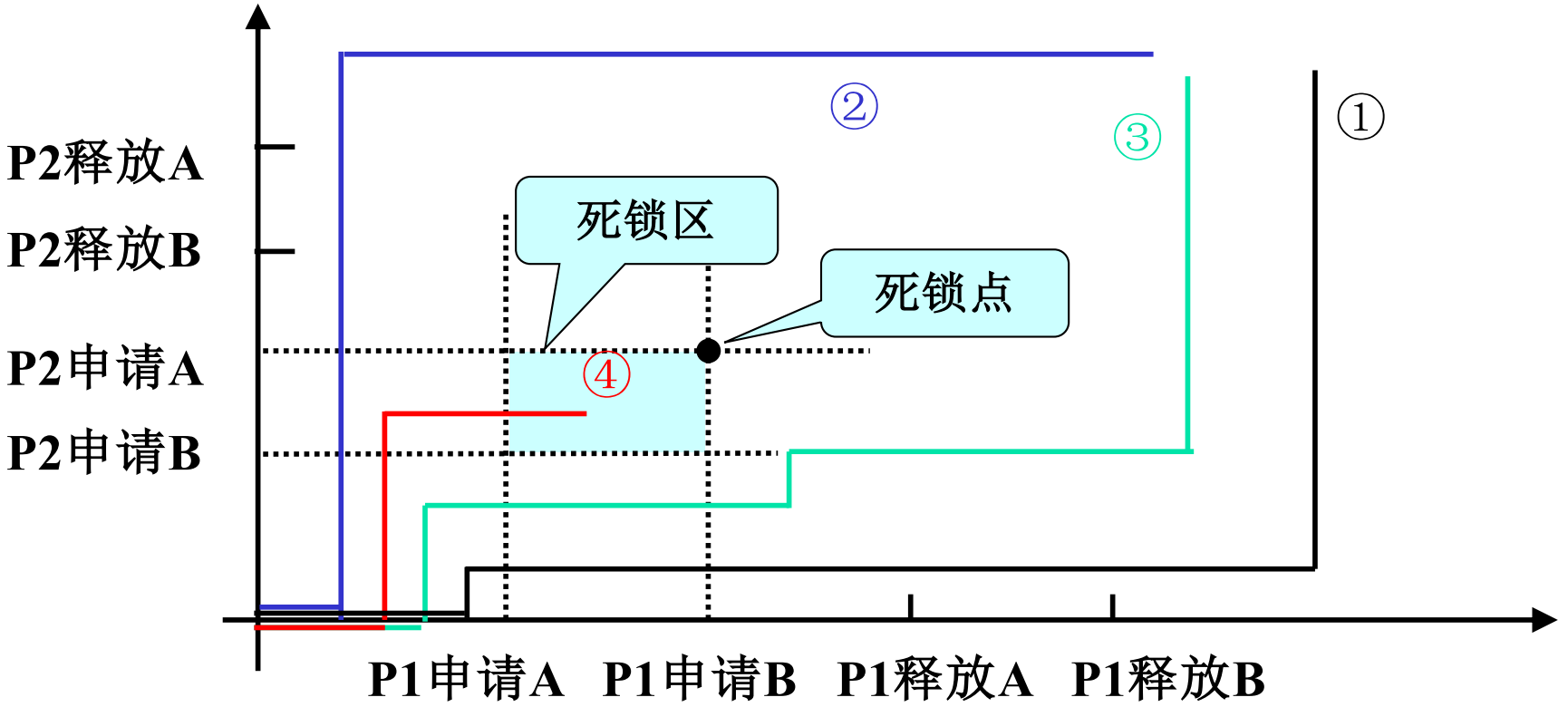
带环且无死锁的资源分配图

死锁产生的原因分析

- 🎯 系统资源的特性
  - 资源分类
    - 可重用资源：使用后可被其他进程使用（CPU、内存）
    - 可消耗资源：使用后消失（信号、消息、中断）
  - 资源使用模式
    - 申请资源 → 使用资源 → 释放资源
- ⚠️ 产生死锁的根本原因
  - 1. 竞争不可抢占资源
    - 进程已获得资源A，还需要资源B
    - 资源B被其他进程占用且不能强制剥夺
  - 2. 进程推进顺序不当
    - 进程申请资源和释放资源的顺序不当导致循环等待

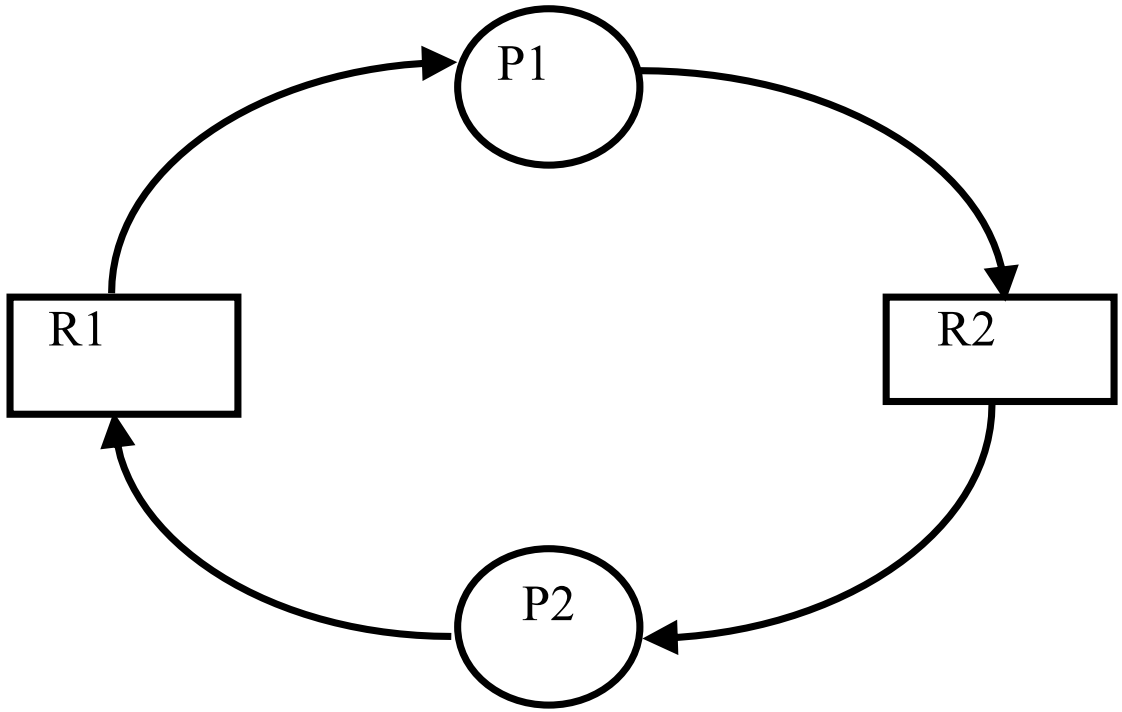
进程推进顺序不当引起的死锁

- 当进程P1、P2共享资源A、B时，若推进顺序合法则不会产生死锁，否则会产生死锁。
- 合法的推进路线：①②③　不合法的推进线路：④



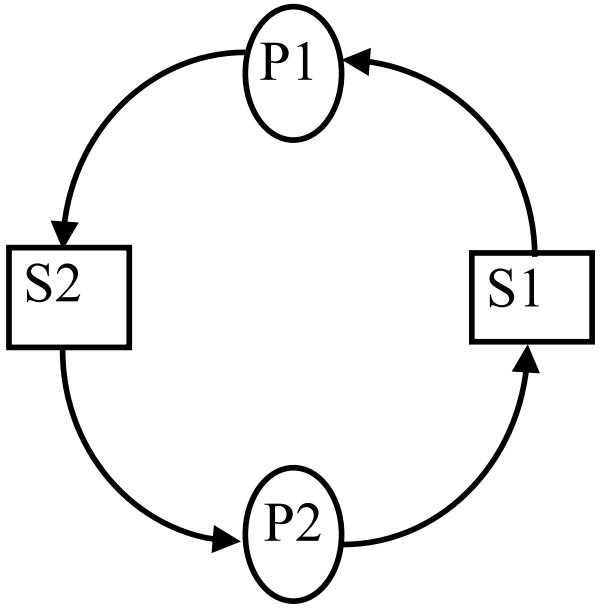
竞争非剥夺资源引起的死锁

- 竞争非剥夺资源例。如，打印机R1和读卡机R2供进程P1和P2共享。



竞争消耗性资源引起的死锁

- 如消息通信按下述顺序进行，则不会发生死锁：
  - P1：...Release(S1)；Request(S2)；...
  - P2：...Release(S2)；Request(S1)；...
- 若按下述顺序，则可能发生死锁：
  - P1：... Request(S2)；Release(S1)；...
  - P2：... Request(S1)；Release(S2)；...

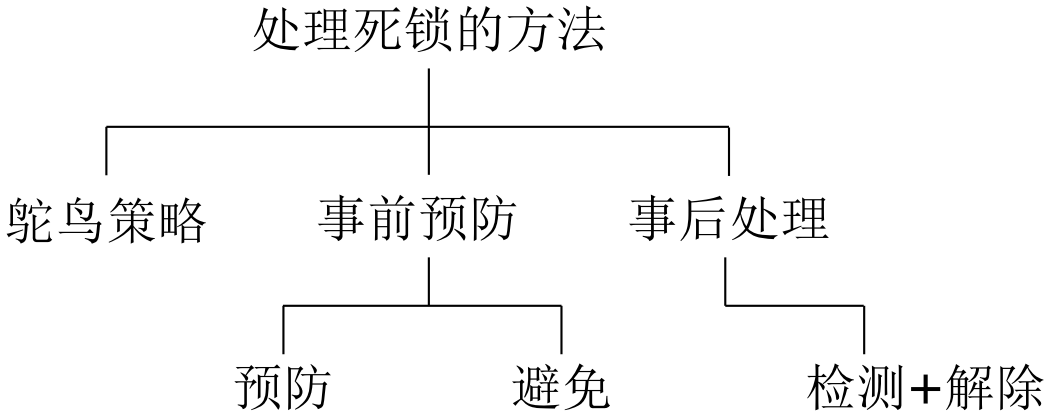


死锁产生的必要条件

- Coffman条件（1971年）：以下四个条件同时满足时，系统可能发生死锁
  - **互斥条件：**
    - 在一段时间内某资源仅为一个进程所占有
    - 例：打印机、写模式的文件
    - why：如果资源可以共享，就不存在竞争
  - **持有并等待：**
    - 进程已持有至少一个资源，同时等待获取其他资源
    - 例：进程持有内存，等待打印机
    - why：如果进程不持有资源，就无法形成循环
  - **资源非抢占：**
    - 资源不能被强制剥夺，只能由持有者主动释放
    - 例子：进程使用的内存不能被强制收回
    - why：如果可以抢占，系统可以打破僵局
  - **循环等待：**
    - 存在一个进程链 P0→P1→...→Pn→P0，每个进程等待下一个进程持有的资源
    - 例：P1等P2的资源，P2等P3的资源，P3等P1的资源
    - why：这是死锁的直接表现
- 互斥 + 持有等待 + 非抢占 + 循环等待 = 死锁
  - 缺少任意一个条件 → 死锁不会发生

死锁的处理策略总览

- 四种基本策略



- 策略对比

策略	时机	代价	资源利用率	适用场景
鸵鸟策略	不处理	极低	高	死锁罕见的系统
死锁预防	系统设计时	高	低	关键系统
死锁避免	运行时决策	中	中	资源需求已知
死锁检测	定期检查	中高	高	批处理系统

策略1：死锁预防

- 核心思想
  - 破坏死锁的四个必要条件之一，使死锁不可能发生
- **方法1：破坏"互斥条件"**
  - 思路：让资源可共享
  - 实现：
    - SPOOLing技术（假脱机）
    - 例子：打印机共享

      进程 → 输出到磁盘缓冲区 → 打印守护进程统一打印
  - 局限性：
    - ❌ 很多资源本质上无法共享（内存、处理器）
    - ❌ 适用范围有限

策略1：死锁预防

- **方法2：破坏"持有并等待"**
  - 思路：进程必须一次性申请所有需要的资源
  - 实现方案A：进程开始前申请全部资源

```
// 方案A：预先分配
acquire_all_resources(R1, R2, R3);  // 一次性获取
use_resources();
release_all_resources(R1, R2, R3);
```

  - 实现方案B：进程申请新资源前必须释放已持有的资源

```
// 方案B：申请前释放
acquire(R1);
// 需要R2时
release(R1);
acquire(R1, R2);  // 重新获取
```

  - 优点：简单有效
  - 缺点：
    - ❌ 资源利用率低（长期持有但不一定用）
    - ❌ 可能导致饥饿（资源需求多的进程难以满足）
    - ❌ 难以预知全部资源需求

策略1：死锁预防

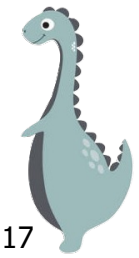
- **方法3：破坏"非抢占"**
  - 思路：允许强制剥夺资源
  - 实现：
    - 当进程申请新资源失败时，释放已持有的所有资源
    - 将释放的资源分配给其他等待进程
    - 只有当进程能重新获得旧资源+新资源时才继续
  - 伪代码：

```
acquire(R1);
if (!acquire(R2)) {   // 获取失败
    release(R1);      // 释放已有资源
    wait();
    retry();
}
```

  - **优点**：资源利用率较高
  - **缺点**：
    - ❌ 只适用于状态易保存的资源（如CPU、内存）
    - ❌ 不适用于打印机等物理设备
    - ❌ 增加系统开销
    - ❌ 可能导致饥饿

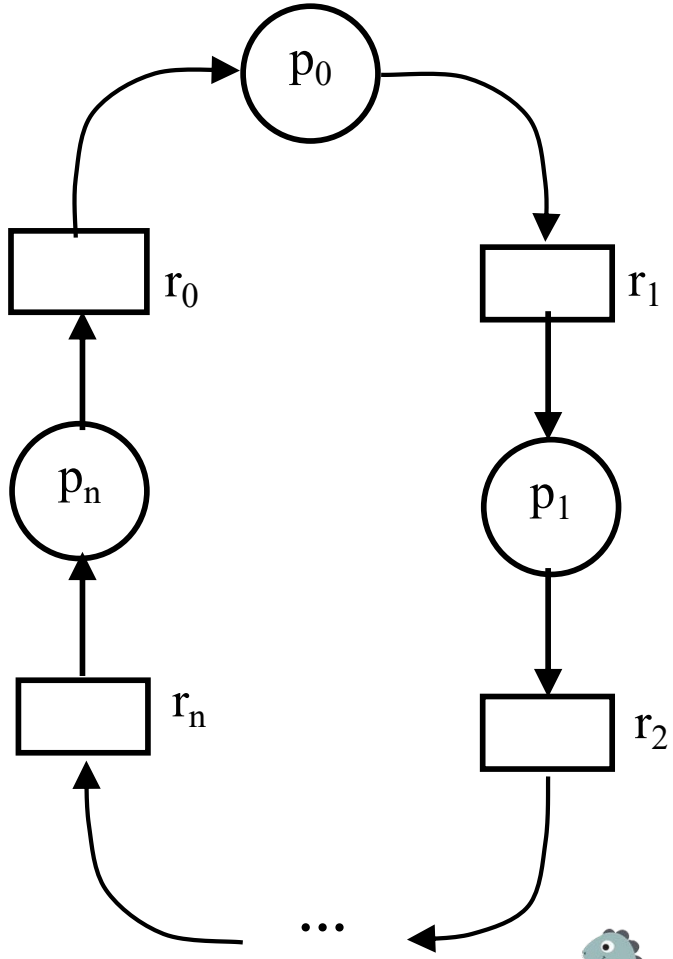
策略1：死锁预防

- **方法4：破坏“循环等待” ⭐推荐**
  - 思路：对资源进行线性排序，按序申请
  - 实现：
    1. 给所有资源编号：R1, R2, ..., Rn
    2. 规定：进程必须按递增顺序申请资源
    3. 已持有Ri的进程只能申请Rj（j > i）
  - 数学证明：可用反证法证明
  - 优点：
    - 不需要预知资源需求
    - 资源利用率较高
    - 实现相对简单
  - 缺点：
    - ❌ 限制了用户编程自由度
    - ❌ 资源编号困难



为什么有序资源分配法可以防止死锁

- 假设循环已经出现并且含于环中的进程是p_0、…、p_n，
- 这意味着p_i正占有r_i类资源，而请求r_{i+1}类资源，
- 设函数f能获得资源序号，则有f（r_i）<f（r_{i+1}），
- 故f（r_0）<f（r_1）<…< f（r_n）<f（r_0）。
- 矛盾，原假设不成立。



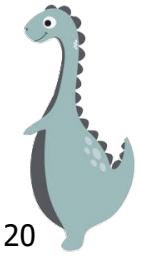
哲学家就餐问题

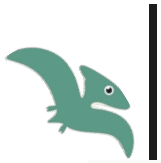
- 经典场景
  - 5个哲学家围坐圆桌 👨👨👨👨👨
  - 5支筷子（每两人之间一支） 🥢🥢🥢🥢🥢
  - 需要同时拿到左右两支筷子才能吃饭
  - 吃完后放下筷子，继续思考
- 当死锁发生时
  - 互斥：资源不能共享（筷子一次只能一人用）
  - 占有并等待：哲学家占有一支筷子，并等待另外一支
  - 非抢占：哲学家不能从别的哲学家手中抢筷子
  - 循环等待：在五个哲学家之间形成了循环等待的情况
- 解决死锁的核心思路：
  - 打破任意一个条件，死锁就不会发生！

方案1：最多4人就餐

```
semaphore room = 4;  // 房间容量
semaphore chopstick[5] = {1, 1, 1, 1, 1};

void philosopher(int i) {
    while(1) {
        think();

        P(room);                        // 进入房间
        P(chopstick[i]);
        P(chopstick[(i+1) % 5]);

        eat();

        V(chopstick[i]);
        V(chopstick[(i+1) % 5]);
        V(room);                        // 离开房间
    }
}
```

- 核心思路
  - 限制同时拿筷子的人数
- 原理：
  - 最多4人持有筷子 → 至少1支筷子空闲
  - → 至少1人能拿到两支筷子 → 可以吃饭
  - → 吃完后释放，其他人可以继续
- 破坏的条件
  - 持有并等待（保证有人不等待）

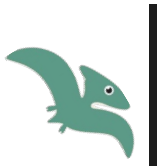




方案2：奇偶策略

```
semaphore chopstick[5] = {1, 1, 1, 1, 1};

void philosopher(int i) {
    while(1) {
        think();

        if (i % 2 == 0) {  // 偶数号：先左后右
            P(chopstick[i]);
            P(chopstick[(i+1) % 5]);
        } else {              // 奇数号：先右后左
            P(chopstick[(i+1) % 5]);
            P(chopstick[i]);
        }

        eat();

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

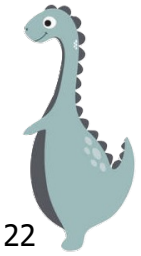
- 核心思路
  - 打破循环等待，规定拿筷子的顺序
- 原理：
  - 偶数号：0→1, 2→3, 4→0（顺时针）
  - 奇数号：2→1, 4→3（逆时针）
  - 0号筷子：只有哲学家0和4争，且0先拿0，4后拿0
  - → 不会形成环路！
- 破坏的条件
  - 循环等待



方案3：原子操作

```
semaphore mutex = 1;  // 保护"拿筷子"操作
semaphore chopstick[5] = {1, 1, 1, 1, 1};

void philosopher(int i) {
    while(1) {
        think();

        P(mutex);                    // 临界区：拿筷子必须原子完成
        P(chopstick[i]);
        P(chopstick[(i+1) % 5]);
        V(mutex);

        eat();

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

- 核心思路
  - 拿筷子必须原子完成，不允许只拿一支
- 原理：
  - 同时只有1人可以拿筷子
  - 要么拿到两支，要么都不拿
  - → 不会出现"拿着一支等另一支"的情况
- 破坏的条件
  - 持有并等待



方案4：状态机（Tanenbaum方案）

- 核心思路：
  - 检查邻居状态，只有两边都不在吃才能拿筷子

```
#define N 5
#define LEFT  (i + N - 1) % N
#define RIGHT (i + 1) % N
#define THINKING 0
#define HUNGRY   1
#define EATING   2
int state[N];                // 每个哲学家的状态
semaphore mutex = 1;         // 保护state数组
semaphore s[N] = {0,0,0,0,0}; // 每个哲学家的私有信号量
void philosopher(int i) {
    while(1) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

```
void take_forks(int i) {
    P(mutex);
    state[i] = HUNGRY;       // 我饿了
    test(i);                 // 尝试拿筷子
    V(mutex);
    P(s[i]);                 // 如果没拿到，就等待
}
void put_forks(int i) {
    P(mutex);
    state[i] = THINKING;
    test(LEFT);              // 看左邻居能不能吃
    test(RIGHT);             // 看右邻居能不能吃
    V(mutex);
}
void test(int i) {
    if (state[i] == HUNGRY &&            // 我饿了
        state[LEFT] != EATING &&         // 左邻居不在吃
        state[RIGHT] != EATING) {        // 右邻居不在吃
        state[i] = EATING;
        V(s[i]);             // 允许自己吃
    }
}
```

策略2：死锁避免

核心思想

- 动态检查资源分配，确保系统始终处于安全状态

关键概念

- 安全状态（Safe State）：
  - 存在一个进程序列 <P1, P2, ..., Pn>，对每个Pi：

    Pi需要的资源 ≤ 当前可用资源 + 所有Pj（j < i）持有的资源
  - 含义：按此序列执行，所有进程都能完成
- 不安全状态（Unsafe State）：
  - 不存在这样的安全序列
  - 注意：不安全 ≠ 死锁，但可能导致死锁

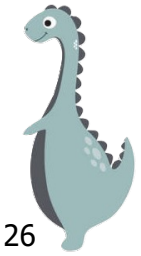
策略2：死锁避免

- 银行家算法（Banker's Algorithm）
  - 算法思想（Dijkstra, 1965）：
    - 银行家不会把所有钱都贷出去
    - 要保留足够资金满足至少一个客户的贷款需求
    - 操作系统像银行家，进程像客户，资源像资金
- 数据结构（n个进程，m种资源）：
  - 关系：Need[i][j] = Max[i][j] - Allocation[i][j]

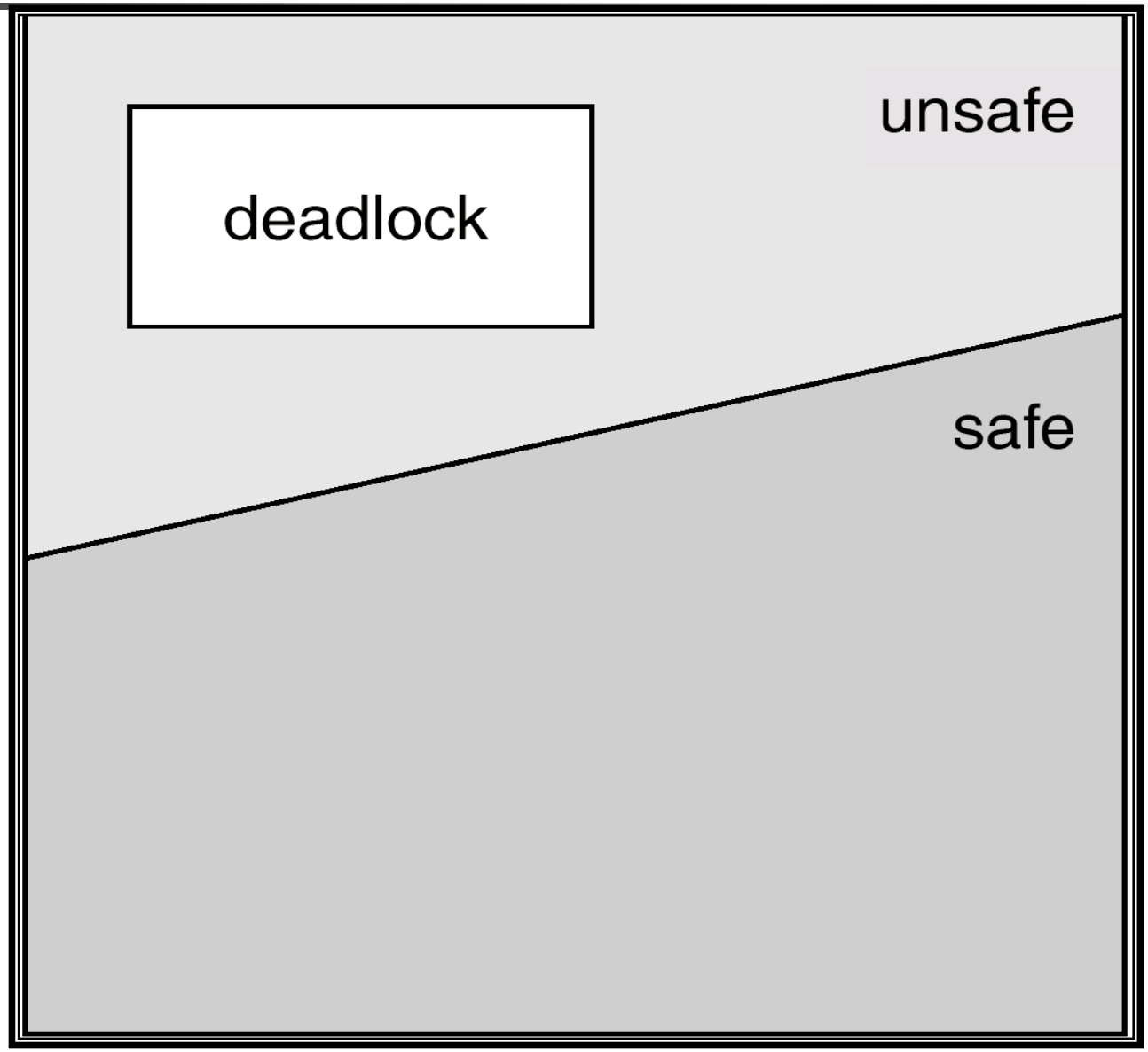
变量	维度	含义
Available	[m]	当前可用资源数量
Max	[n][m]	每个进程的最大资源需求
Allocation	[n][m]	每个进程已分配的资源
Need	[n][m]	每个进程还需要的资源

安全状态

- 安全状态是指系统能按某种顺序如< P_1、P_2... 、P_n>来为每个进程分配其所需的资源，直至最大需求，使每个进程都可以顺利完成，则称此时的系统状态为安全状态，称序列< P_1、P_2、...、P_n >为安全序列。
- 若不存在一个安全序列，则称系统状态为不安全状态。
- 安全状态不是死锁状态，死锁状态是不安全状态



安全、不安全、死锁状态空间

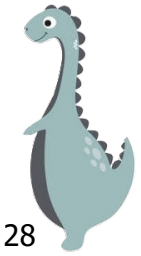


安全状态例

- At time T_0，there are 3 free.

	Max Needs	Current Needs	free
P0	10	5	3
P1	4	2	
P2	9	7	

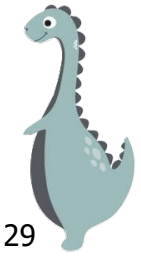
这时P1可立即得到需要的所有资源，并能将资源归还。



安全状态例

	Max Needs	Current Needs	free
P0	10	5	5
P1	4	2	
P2	9	7	

- 接着P0可得到需要的所有资源，并能将资源归还。



安全状态例

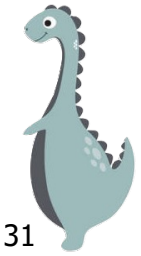
	Max Needs	Current Needs	free
P0	10	5	10
P1	4	2	
P2	9	7	

- 最后P2可得到需要的所有资源，并能将资源归还。
- 在T0时刻，系统状态安全。序列<P1、P0、P2>满足安全条件。

从安全状态转换为不安全状态

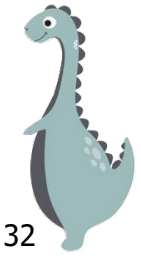
- 假设在T1时刻，进程P2申请并得到了1个资源，系统就不安全了。

	Max Needs	Current Needs	free
P0	10	5	2
P1	4	2	
P2	9	6	



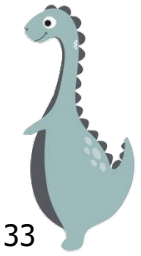
银行家算法

- 最具代表性的死锁避免算法是Dijkstra的银行家算法。
- 每一个进程必须事先声明资源最大使用量
- 当用户申请一组资源时，系统必须确定这些资源的分配是否仍会使系统处于安全状态，如果是就可分配资源，否则等待。



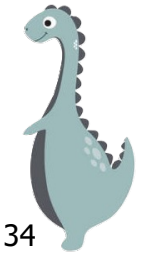
银行家算法的数据结构

- 设n为进程的数目，m为资源类型的数目
- 可用资源向量Available：长度为m的向量。表示每种资源的现有实例数。
  - 如果available[j]=k, 那么R_j类资源现在有k个实例。
- 最大需求矩阵Max：n×m的矩阵，定义了每个进程的最大需求。
  - 如果Max[i,j]=k,那么进程P_i最多可以请求R_j类资源的k个实例



银行家算法的数据结构

- 分配矩阵Allocation： n×m 的矩阵。定义每个进程现在所分配的各类资源实例数目。
  - 如果Allocation[i,j]=k,那么进程P_i当前分配了R_j类资源的k个实例。
  - Allocationi表示进程P_i的分配向量，由矩阵Allocation的第i行构成。
- 需求矩阵Need：n×m 的矩阵，表示每个进程还需要的资源数目。
  - 如果Need(i，j)＝k，表示进程P_i还需R_j类资源k个。
  - $Need_i$表示进程P_i的需求向量，由矩阵Need的第i行构成。
  - Need [i,j] = Max[i,j] – Allocation [i,j].



需求矩阵

- Need：n×m 的矩阵，表示每个进程还需要的资源数目。

  Need: An n×m matrix indicates the remaining resource need of each process.

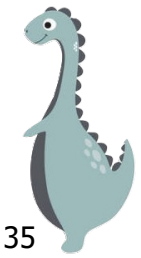
- 如果Need(i，j)＝k，表示进程P_i还需R_j类资源k个。

  If Need[i,j] = k, then P_i may need k more instances of R_j to complete its task.

- $Need_i$表示进程P_i的需求向量，由矩阵Need的第i行构成。

- Note that:

  Need [i,j] = Max[i,j] – Allocation [i,j].



安全性算法 Safety Algorithm

- 1）设Work和Finish分别是长度为m和n的向量。初始化为：
  - Work=available
  - Finish[i]=false　i=1,2, … ,n
- 2）查找这样的进程i使其满足：
  - Finish(i)= false ；
  - $Need_i \leq Work$ ；
  - 如果没有这样的i存在，则进入步骤4)
- 3）当进程Pi获得资源后，可顺利执行直到完成，并释放出分配给它的资源，故应执行：
  - $Work=Work+Allocation_i$
  - Finish[i]=true
  - 转步骤2)
- 4）如果对所有i，Finish[i]=true，那么系统处于安全状态，否则处于不安全状态

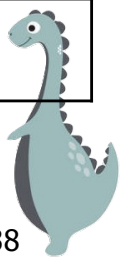
资源请求算法（银行家算法）

- 设$Request_i$是进程P_i的请求向量，如$Request_i(j)=k$，那么进程P_i需要R_j类资源k个。
- 当P_i发出资源请求后，系统按下述步骤进行检查
  - 1）如果$Request_i \le Need_i$，则转向步骤2；否则出错。
  - 2）如果$Request_i \le Available$，则转向步骤3；否则P_i等待。
  - 3）试分配并修改数据结构：假设分配P_i请求的资源
    - **$Available = Available - Request_i$；**
    - **$Allocation_i = Allocation_i + Request_i$；**
    - **$Need_i = Need_i - Request_i$；**
  - 4）系统执行安全性算法，检查此次资源分配是否安全。若安全，才正式分配；否则，试分配作废，让进程P_i等待。

银行家算法例

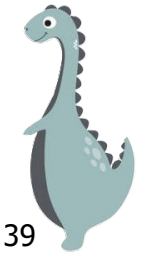
- 假定系统中有5个进程 P0、P1、P2、P3、P4和三种类型的资源A、B、C，数量分别为12、5、9，在T0时刻的资源分配情况如下所示。

资源情况 / 进程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	8	5	3	1	1	0	7	4	3	3	3	2
P_1	3	2	3	2	0	1	1	2	2			
P_2	9	0	3	3	0	3	6	0	0			
P_3	2	2	2	2	1	1	0	1	1			
P_4	5	3	3	1	0	2	4	3	1			



T_0时刻的安全性

- 利用安全性算法对T_0时刻的资源分配情况进行分析，可得如下所示的T_0时刻的安全性分析。



T_0时刻的安全性检查

Need0：7,4,3　Need1：1,2,2　Need2：6,0,0　Need3：0,1,1　Need4：4,3,1
Alloc0：1,1,0　Alloc1：2,0,1　Alloc2：3,0,3　Alloc3：2,1,1　Alloc4：1,0,2
Avail 332

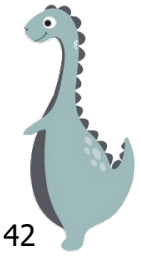
资源情况 / 进程	Work	Need	Alloc	Work+Alloc	Finish
	A B C	A B C	A B C	A B C	
P_1	3 3 2	1 2 2	2 0 1	5 3 3	true
P_3	5 3 3	0 1 1	2 1 1	7 4 4	true
P_4	7 4 4	4 3 1	1 0 2	8 4 6	true
P_2	8 4 6	6 0 0	3 0 3	11 4 9	true
P_0	11 4 9	7 4 3	1 1 0	12 5 9	true

T_0时刻是安全的

- 从上述分析得知，T_0时刻存在着一个安全序列< P_1、P_3、P_4、P_2、P_0 >，故系统是安全的。

P_1请求资源

- P_1发出请求向量$Request_1$（1，0，2），系统按银行家算法进行检查：
  - 1）$Request_1$(1，0，2) ≤$Need_1$(1，2，2)
  - 2）$Request_1$(1，0，2) ≤ Available(3，3，2)
  - 3）系统先假定可为P_1分配资源，并修改Available、$Allocation_1$、$Need_1$向量，由此形成的资源变化情况如下所示。



为P_1试分配资源后

资源情况 / 进程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	8	5	3	1	1	0	7	4	3	2	3	0
P_1	3	2	3	3	0	3	0	2	0			
P_2	9	0	3	3	0	3	6	0	0			
P_3	2	2	2	2	1	1	0	1	1			
P_4	5	3	3	1	0	2	4	3	1			

- 4）再利用安全性算法检查此时系统是否安全，可得如下所示的安全性分析。

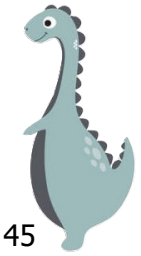
P_1申请资源后的安全性检查

Need0：7,4,3　Need1：0,2,0　Need2：6,0,0　Need3：0,1,1　Need4：4,3,1
Alloc0：1,1,0　Alloc1：3,0,3　Alloc2：3,0,3　Alloc3：2,1,1　Alloc4：1,0,2
Avail 230

资源情况 / 进程	Work	Need	Alloc	Work+Alloc	Finish
	A B C	A B C	A B C	A B C	
P_1	2 3 0	0 2 0	3 0 3	5 3 3	true
P_3	5 3 3	0 1 1	2 1 1	7 4 4	true
P_4	7 4 4	4 3 1	1 0 2	8 4 6	true
P_2	8 4 6	6 0 0	3 0 3	11 4 9	true
P_0	11 4 9	7 4 3	1 1 0	12 5 9	true

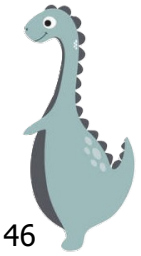
可以为P_1分配资源

- 从上述分析得知，可以找到安全序列<P_1、P_3、P_4、P_0、P_2>，系统安全，可以分配。



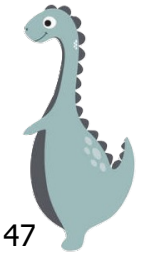
P_4请求资源

- P_4发出请求向量$Request_4$(3，3，0)，系统按银行家算法进行检查：
  - 1）$Request_4$(3，3，0) ≤$Need_4$(4，3，1)
  - 2）$Request_4$(3，3，0)＞ Available(2，3，0)，让P4等待。



P_0请求资源

- P_0发出请求向量$Request_0$ (0，2，0)，系统按银行家算法进行检查：
  - 1）$Request_0(0，2，0) \leq Need_0(7，4，3)$
  - 2）$Request_0(0，2，0) \leq Available(2，3，0)$
  - 3）系统先假定可为P_0分配资源，并修改有关数据，如下所示。



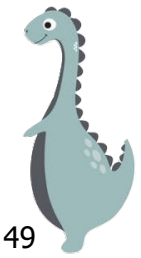
为P_0试分配资源后

资源情况 / 进程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	8	5	3	1	3	0	7	2	3	2	1	0
P1	3	2	3	3	0	3	0	2	0			
P2	9	0	3	3	0	3	6	0	0			
P3	2	2	2	2	1	1	0	1	1			
P4	5	3	3	1	0	2	4	3	1			

- 4）再利用安全性算法检查此时系统是否安全。从上表中可以看出，可用资源Available（2，1，0）已不能满足任何进程的需要，故系统进入不安全状态，此时系统不分配资源。

策略2：死锁避免

- 优点：
  - 不需要破坏任何死锁必要条件
  - 资源利用率高
  - 允许进程动态申请资源
- 缺点：
  - ✗ 需要预知最大资源需求（不现实）
  - ✗ 进程数固定
  - ✗ 算法开销大（$O(n^2m)$）
  - ✗ 实际系统很少使用



策略3：死锁检测

核心思想

- 允许死锁发生，定期检测，发现后解除

检测算法

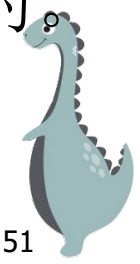
- 资源分配图简化（单资源实例）：
  - 进程节点 + 资源节点
  - 存在环 → 死锁

检测时机

- 策略选择：
  - 每次资源分配时检测：开销大，但能快速发现
  - 定时检测：平衡开销与响应时间
  - CPU利用率下降时检测：可能发生了死锁

死锁判定法则

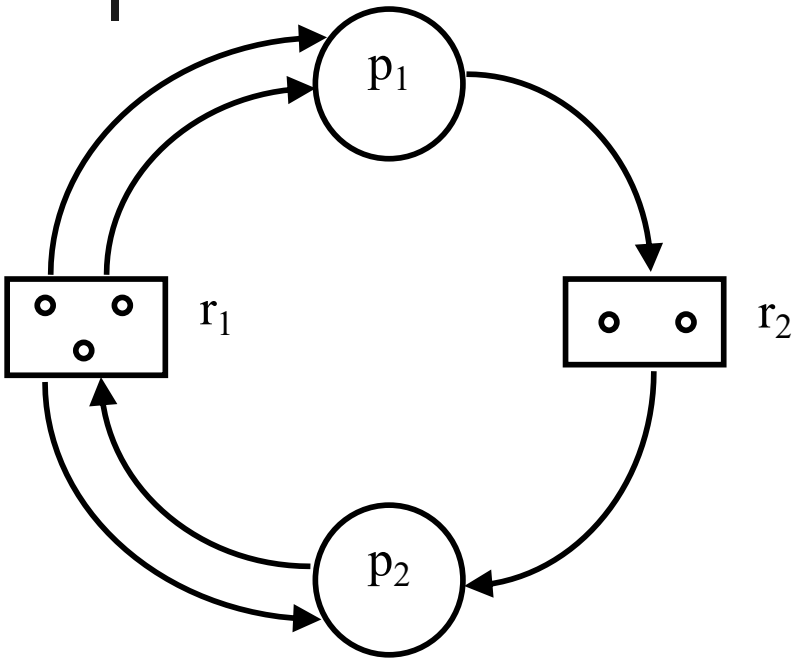
- 将资源分配图简化可以检测系统状态S是否为死锁状态，方法如下：
  - 在资源分配图中，找出一个既不阻塞又非孤立的进程结点p_i，进程p_i获得了它所需要的全部资源，能运行完成然后释放所有资源。这相当于消去p_i的所有请求边和分配边，使之成为孤立结点。
  - 进程p_i释放资源后，可以唤醒因等待这些资源而阻塞的进程，从而可能使原来阻塞的进程变为非阻塞进程。
  - 在进行一系列化简后，若能消去图中所有的边，使所有进程都成为孤立结点，则称该图是可完全简化的；若不能使该图完全化简，则称该图是不可完全简化的。

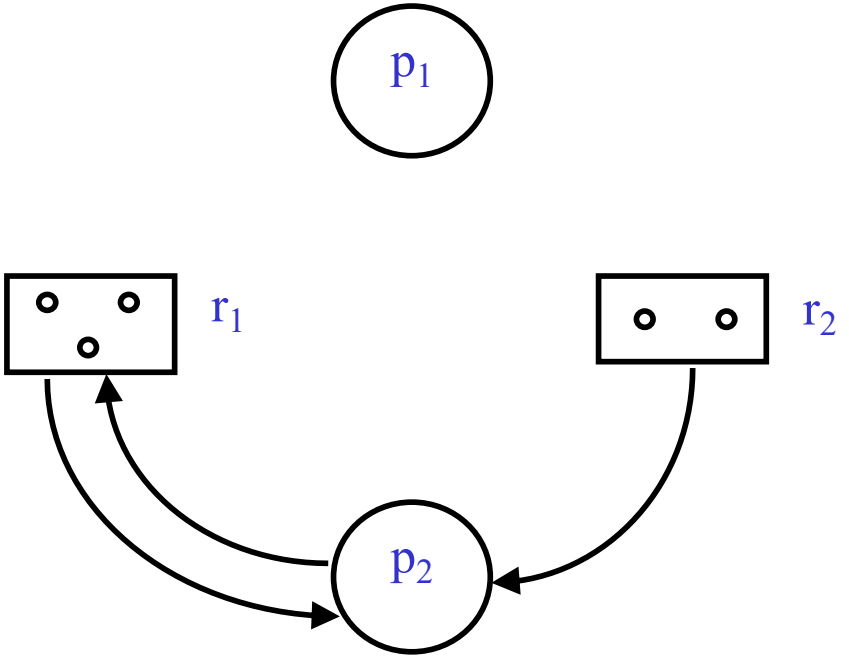


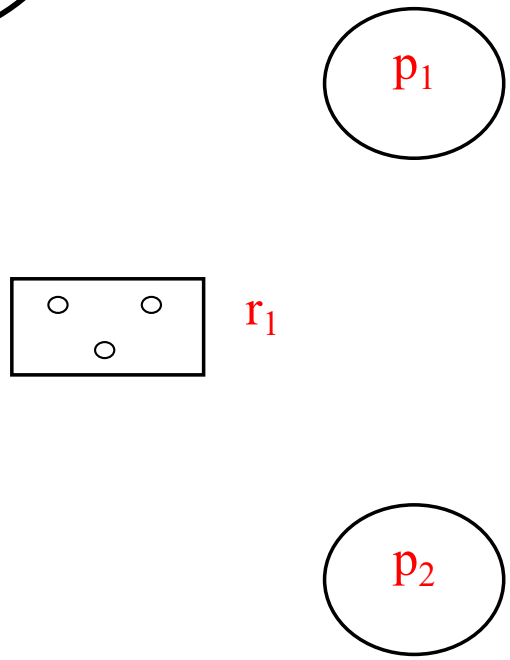
死锁定理

- 可以证明：所有的简化顺序将得到相同的不可简化图。
- S为死锁状态的条件是当且仅当S状态的资源分配图是不可完全简化的。该条件称为死锁定理。

资源分配图简化例

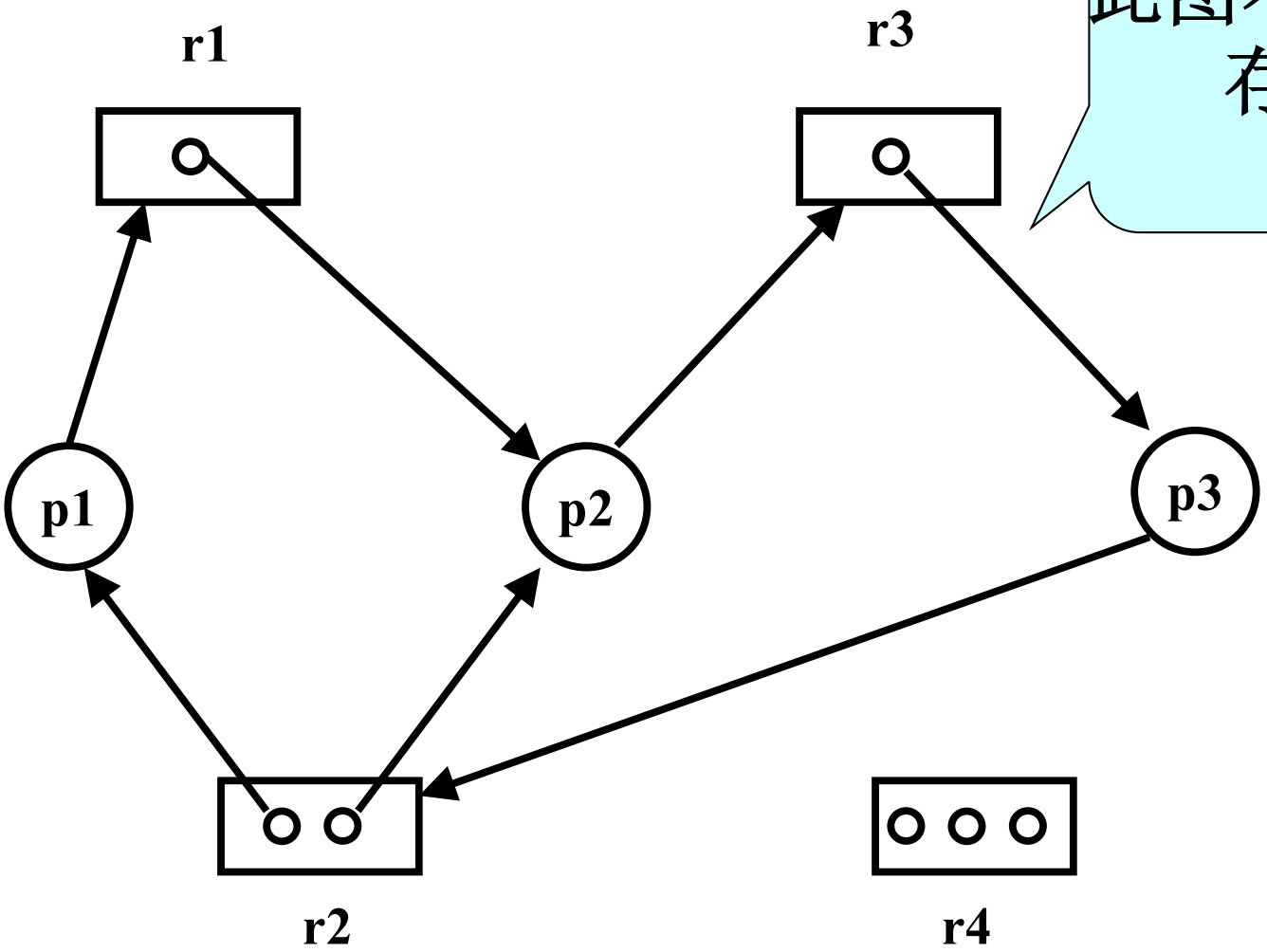






资源分配图简化例

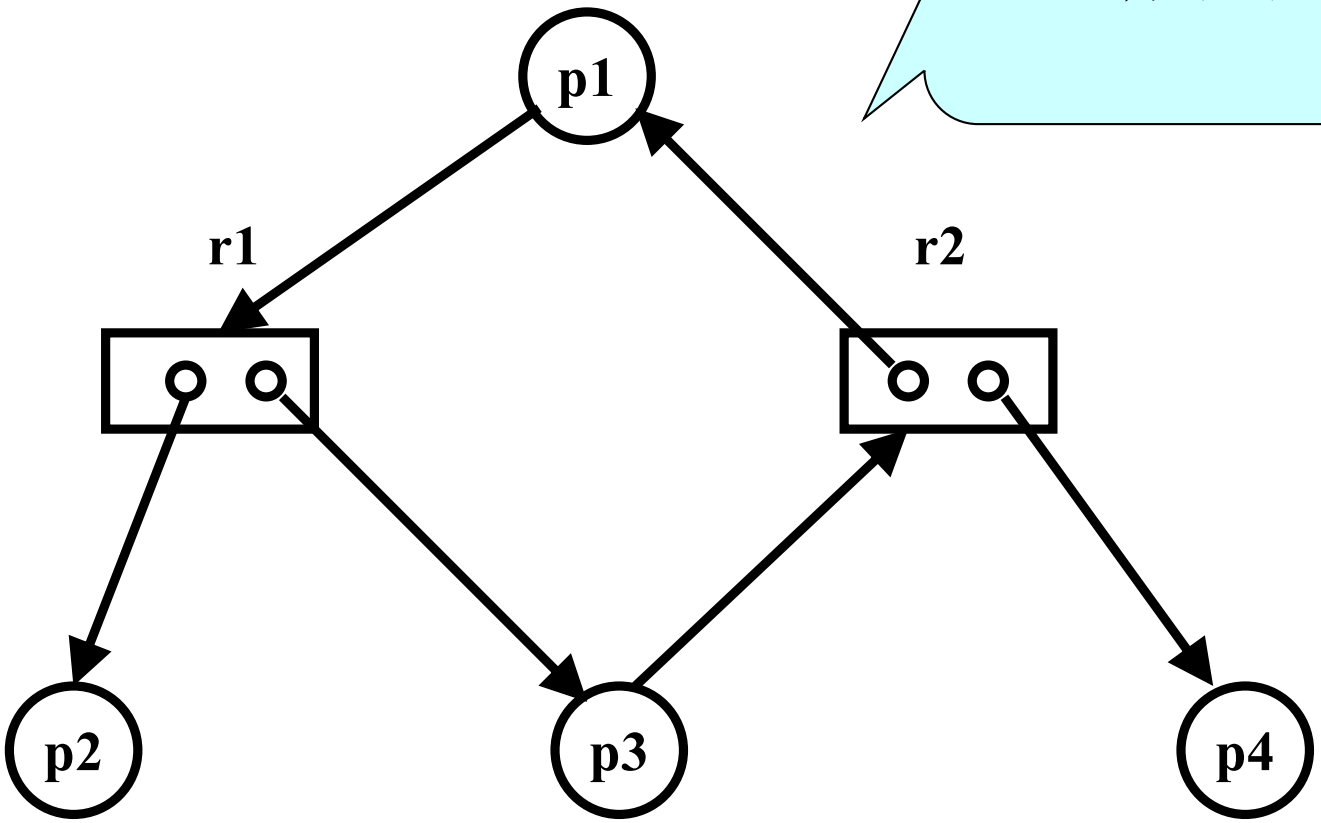
- 下图是否存在死锁？



此图不可完全简化，存在死锁。

资源分配图简化例

- 下图是否存在死锁？



此图可以完全简化，不存在死锁。

死锁检测算法中的数据结构

- Available：长度为m的向量，表示每类资源的可用数目。
- Allocation： n×m矩阵，表示当前每个进程已分配的资源数目。
- 请求矩阵Request： n×m矩阵，表示当前每个进程的资源请求数目。
- 工作向量Work：表示系统当前可提供资源数。
- 进程集合L：记录当前已不占用资源的进程。

死锁检测的算法

Work=Available;

$L=<L_i \mid Allocation_i=0 \cap Request_i=0>$

for all $L_i \notin L$ do

{

if（$Request_i \leq Work$）

{

$Work=Work+Allocation_i$;

$L=L \cup L_i$;

}

}

$deadlock=\neg(L==<p_1,p_2,\ldots p_n>)$

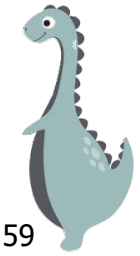
策略4：死锁解除

- 检测到死锁后如何恢复？

- 方法1：进程终止
  - 方案A：终止所有死锁进程
    - 优点：简单直接
    - 缺点：代价大，已完成工作丢失
  - 方案B：逐个终止直到死锁解除
    - 每次终止一个进程
    - 重新检测是否解除
    - 重复直到死锁消失
  - 终止顺序考虑因素：
    - 1. 进程优先级
    - 2. 已运行时间
    - 3. 已使用资源
    - 4. 还需要的资源
    - 5. 进程类型（交互/批处理）

策略4：死锁解除

- 检测到死锁后如何恢复？
  - 方法2：资源抢占
  - 步骤：
    1. 选择牺牲者：选择代价最小的进程
    2. 回滚：返回到安全状态，重新执行
    3. 防止饥饿：限制同一进程被抢占次数
  - 代价评估：
    - 持有资源的数量
    - 已执行时间
    - 还需执行时间



实际系统中的策略

系统	策略	原因
Linux	鸵鸟策略+部分预防	死锁极少，处理代价高
Windows	鸵鸟策略+超时机制	实用主义
数据库系统	检测+解除	事务可回滚
嵌入式系统	预防	可预测性重要

⚖️ 策略选择考虑因素

1. 死锁发生频率：频繁则需要预防，罕见则可忽略
2. 系统类型：实时系统必须预防，批处理可检测
3. 资源类型：物理资源难抢占，逻辑资源易回滚
4. 性能要求：高性能系统倾向于检测
5. 开发成本：预防需要仔细设计

- 🎯 混合策略（最常见）—— 不同资源采用不同策略
  - 内存：预防（线性排序）
  - 磁盘空间：预防（预分配）
  - 数据库锁：检测+解除
  - I/O设备：避免（请求次序）

练习题

- 8.3 Consider the following snapshot of a system:

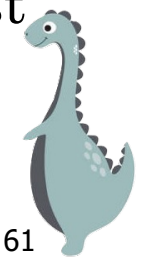
	Allocation	Max	Available
	A B C D	A B C D	A B C D
T_0	0 0 1 2	0 0 1 2	1 5 2 0
T_1	1 0 0 0	1 7 5 0	
T_2	1 3 5 4	2 3 5 6	
T_3	0 6 3 2	0 6 5 2	
T_4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix Need?

b. Is the system in a safe state?

c. If a request from thread T1 arrives for (0,4,2,0), can the request be granted immediately



练习题

8.18 Which of the six resource-allocation graphs shown in Figure 8.12 illustrate deadlock? For those situations that are deadlocked, provide the cycle of threads and resources. Where there is not a deadlock situation, illustrate the order in which the threads may complete execution.

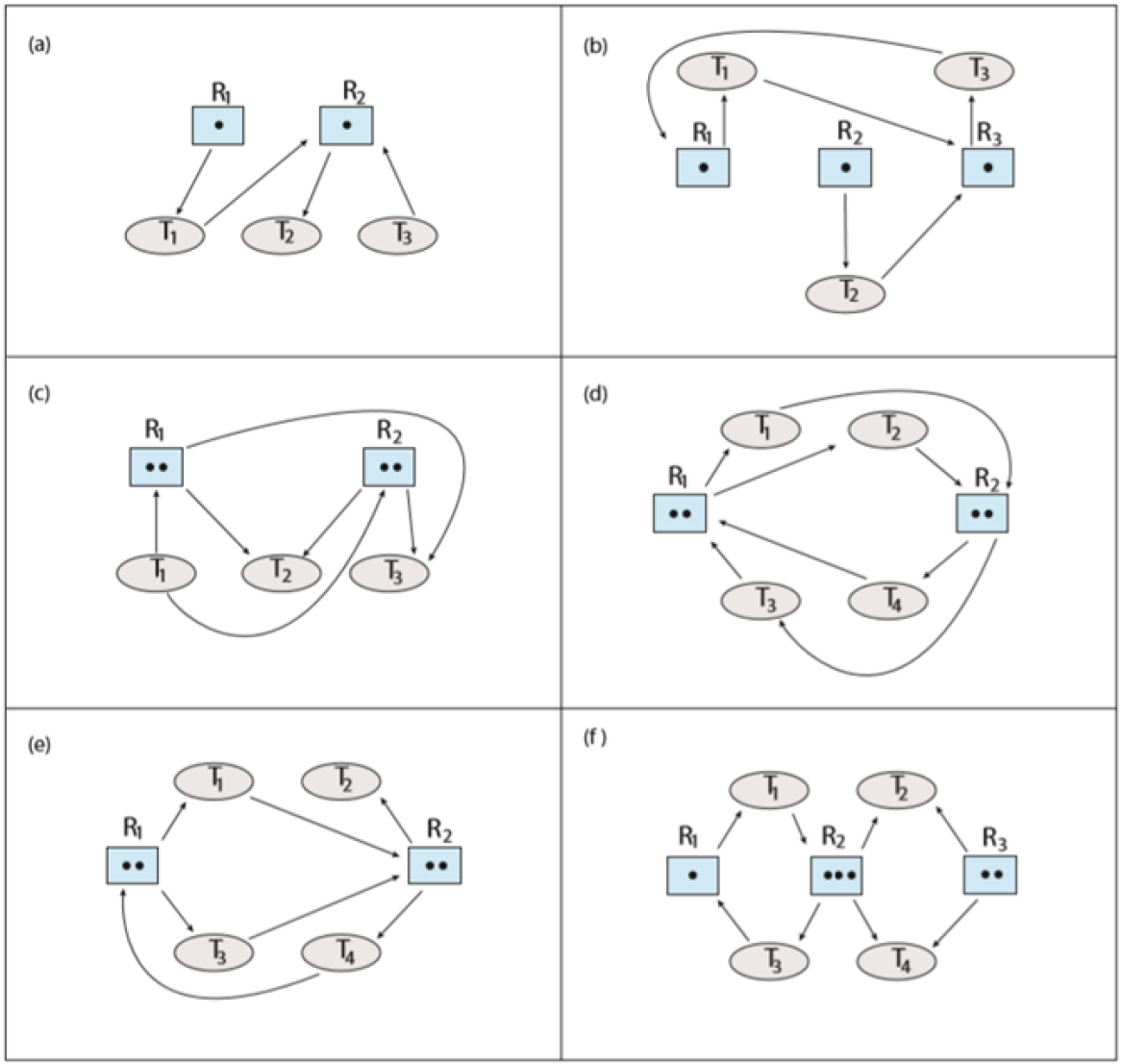


Figure 8.12 Resource-allocation graphs for Exercise 8.18.

选择题1

- 要防止死锁的发生，可以通过破坏这四个必要条件之一来实现，但破坏 _____ 条件是不太实际的。

  A.循环等待　　B.部分分配

  C.不可抢占　　D.互斥

- 为多道程序提供的可共享资源不足时，可能出现死锁。但是，不适当的 _____ 也可能产生死锁。

  A.分配队列优先权　　B.进程推进顺序

  C.资源的线性分配　　D.进程优先权

选择题2

- 采用资源剥夺法可以解除死锁，还可以采用 _____ 方法解除死锁。

  A.拒绝分配新资源　　B.修改信号量

  C.执行并行操作　　D.撤消进程

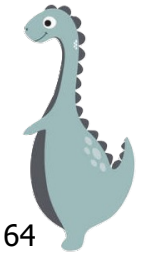
- 在 _____ 的情况下，系统出现死锁。

  A. 计算机系统发生了重大故障

  B. 有多个封锁的进程同时存在

  C. 若干进程因竞争资源而无休止地相互等待他方释放已占有的资源

  D. 资源数大大小于进程数或进程同时申请的资源数大大超过资源总数



选择题3

- 银行家算法在解决死锁问题中是用于 _____ 的。

  A.检测死锁　　B.预防死锁

  C.避免死锁　　D. 解除死锁

- 资源的有序分配策略可以破坏 _____ 条件。

  A.占有且等待资源　　B.循环等待资源

  C. 非抢夺资源　　D.互斥使用资源

选择题4

- 某系统中有3个并发进程，都需要同类资源4个，试问该系统不会发生死锁的最少资源数是_____。

  A. 9　　B. 10

  C. 11　　D. 12

- 有序资源分配方法属于_____方法。

  A．死锁预防　　B．死锁避免

  C．死锁检测　　D．死锁解除

选择题5

- 不能防止死锁的资源分配策略是_____。

  A．剥夺式分配方式　B．按序分配方式

  C．静态分配方式　D．互斥使用分配方式

- 某计算机系统中有6台打印机，多个进程均最多需要2台打印机，规定每个进程一次仅允许申请一台打印机。为保证一定不发生死锁，则允许参与打印机资源竞争的最大进程数是__。

  A．3　B．4　C．5　D．6

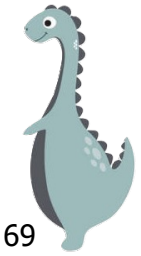
填空题

- 在有m个进程的系统中出现死锁时，死锁进程的个数k应该满足的条件是_____。
- 银行家算法中，当一个进程提出的资源请求将导致系统从 ① 进入 ② 时，系统就拒绝它的资源请求。
- 对待死锁，一般应考虑死锁的预防、避免、检测和解除四个问题。典型的银行家算法是属于 ①，破坏环路等待条件是属于 ②，而剥夺资源是 ③ 的基本方法。

考研题1

- 某计算机中有8台打印机，由K个进程竞争使用，每个进程最多需要3台打印机。该系统可能会发生死锁的K的最小值为（）。09

  A、2　B、3　C、4　D、5



考研题2

- 某时刻进程的资源使用情况如下表所示：

进 程	已分配资源			尚需资源			可用资源		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	2	0	0	0	0	1	0	2	1
P2	1	2	0	1	3	2			
P3	0	1	1	1	3	1			
P4	0	0	1	2	0	0			

- 此时的安全序列是___。11

A、P1，P2，P3，P4　　B、P1，P3，P2，P4

C、P1，P4，P3，P2　　D、不存在

考研题3

- 假设5个进程P0，P1，P2，P3，P4共享三类资源R1、R2、R3，这些资源总数分别为18，6，22。T0时刻的资源分配情况如下表所示，此时存在一个安全序列是：12

	已分配			最大需求		
	R1	R2	R3	R1	R2	R3
P0	3	2	3	5	5	10
P1	4	0	3	5	3	6
P2	4	0	5	4	0	11
P3	2	0	4	4	2	5
P4	3	1	4	4	2	4

A. P0,P2,P4,P1,P3

B. P1,P0,P3,P4,P2

C. P2,P1,P0,P3,P4

D. P3,P4,P2,P1,P0