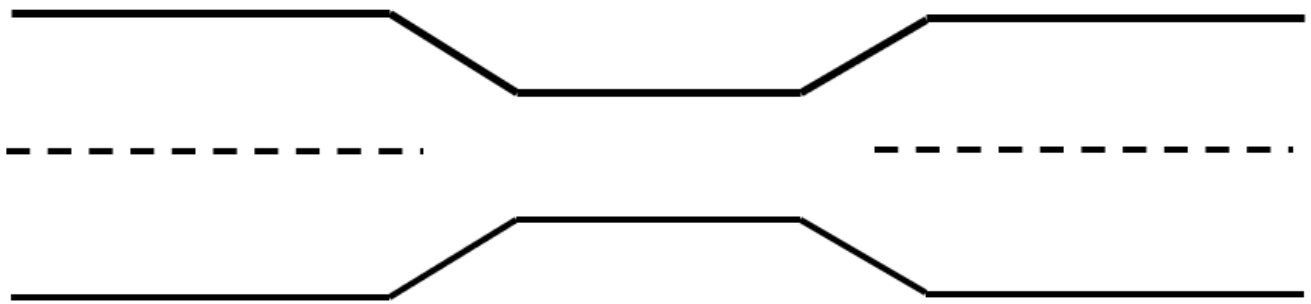
第十二讲 同步与互斥

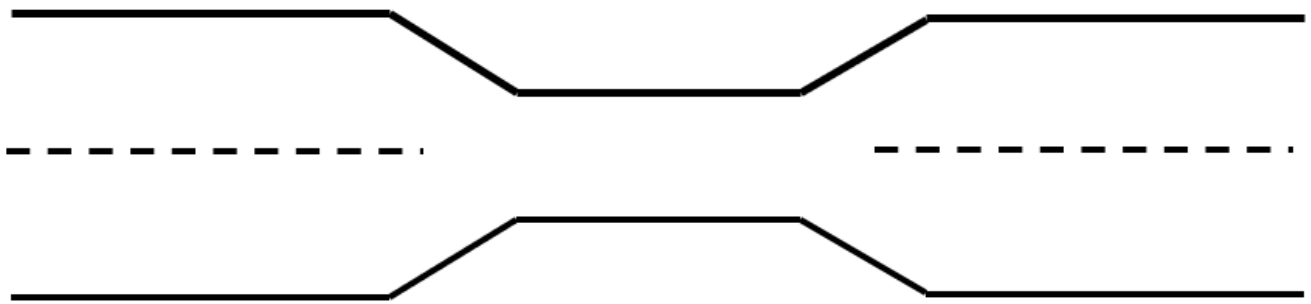
第五节 死锁

死锁问题



- 桥梁只能单向通行
- 桥的每个部分可视为一个资源
- 可能出现死锁
  - 对向行驶车辆在桥上相遇
  - 解决方法：一个方向的车辆倒退(资源抢占和回退)

死锁问题

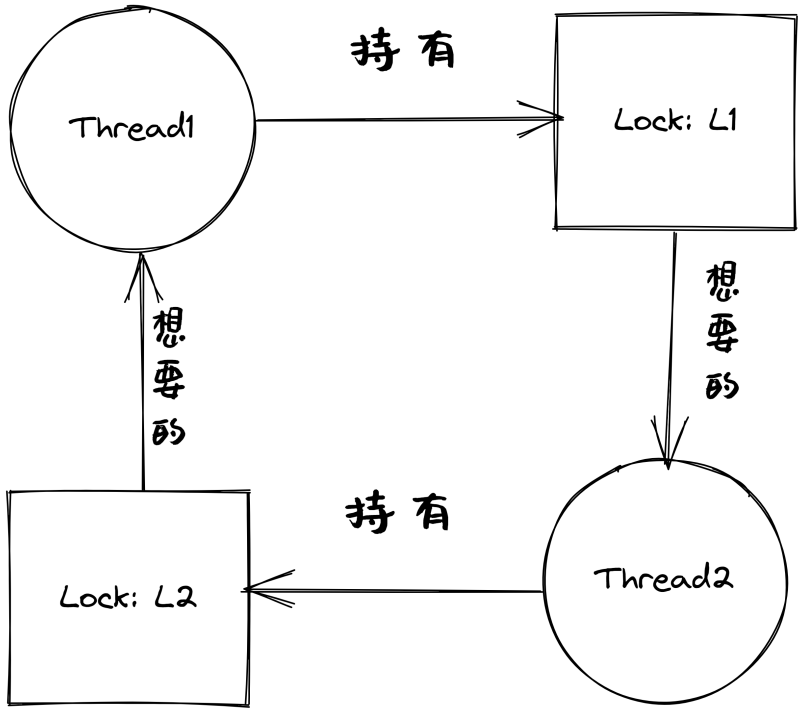


- 桥梁只能单向通行
- 桥的每个部分可视为一个资源
- 可能发生饥饿
  - 由于一个方向的持续车流，另一个方向的车辆无法通过桥梁

死锁问题

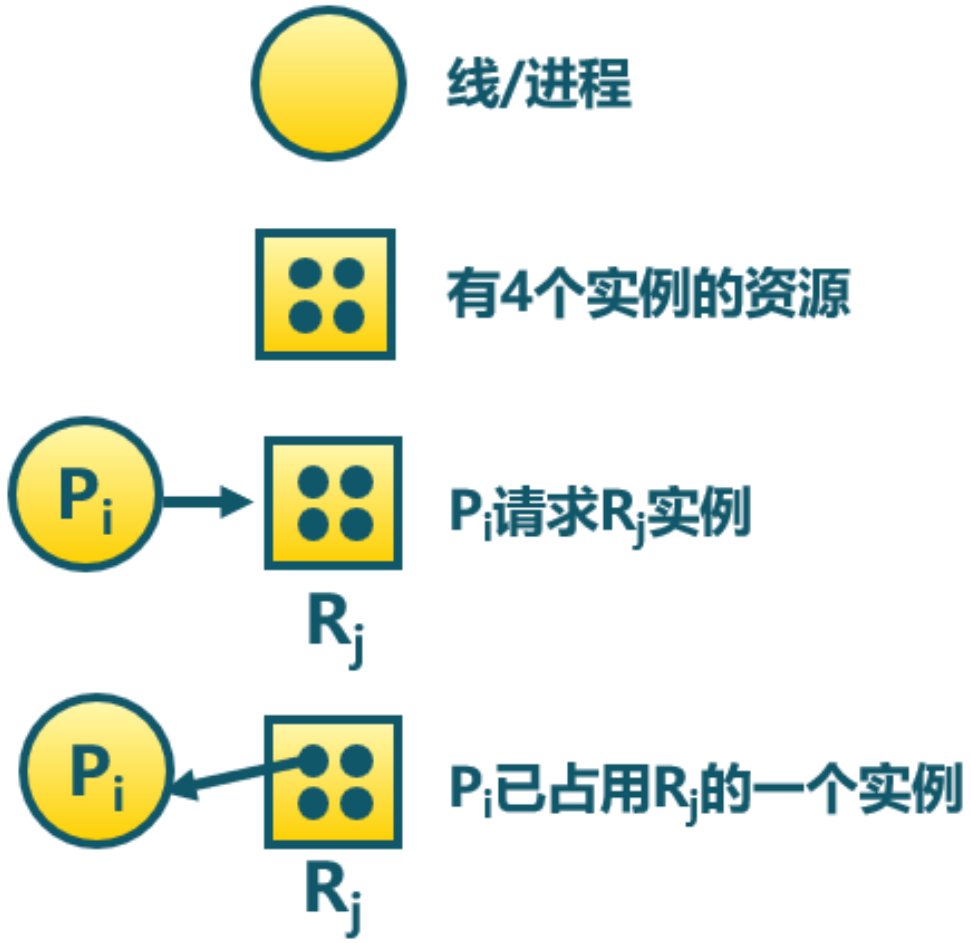
由于竞争资源或者通信关系，两个或更多线程在执行中出现，永远相互等待只能由其他进程引发的事件

```
Thread 1:    Thread 2:
lock(L1);    lock(L2);
lock(L2);    lock(L1);
```



死锁问题 -- 资源

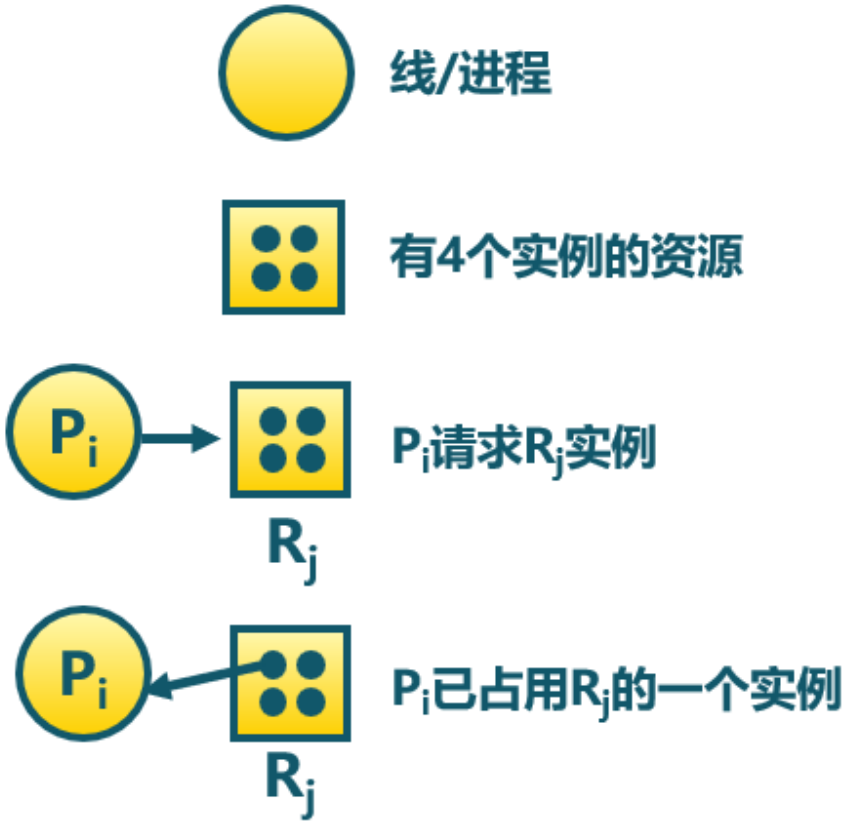
- 资源类型$R_1, R_2, \ldots, R_m$
  - CPU执行时间、内存空间、I/O设备等
- 每类资源R_i有W_i个实例
- 线/进程访问资源的流程
  - 请求：申请空闲资源
  - 使用：占用资源
  - 释放：资源状态由占用变成空闲



死锁问题 -- 资源

资源分类

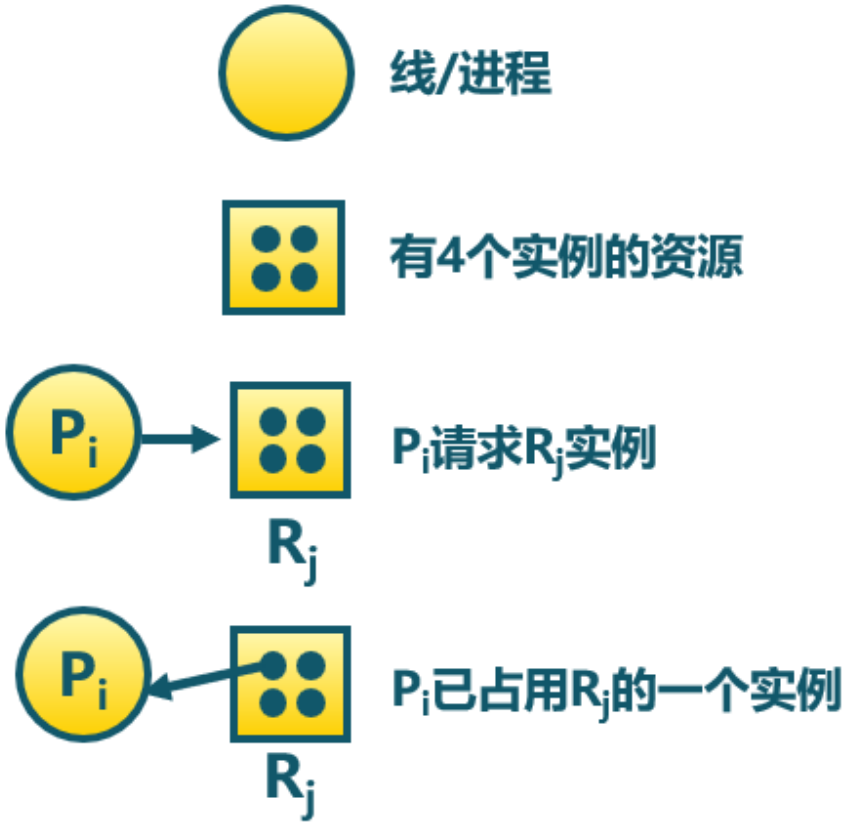
- 可重用资源（Reusable Resource）
  - 任何时刻只能有一个线/进程使用资源
  - 资源被释放后，其他线/进程可重用
  - 可重用资源示例
    - 硬件：处理器、内存、设备等
    - 软件：文件、数据库和信号量等
  - 可能出现死锁：每个进程占用一部分资源并请求其它资源



死锁问题 -- 资源

资源分类

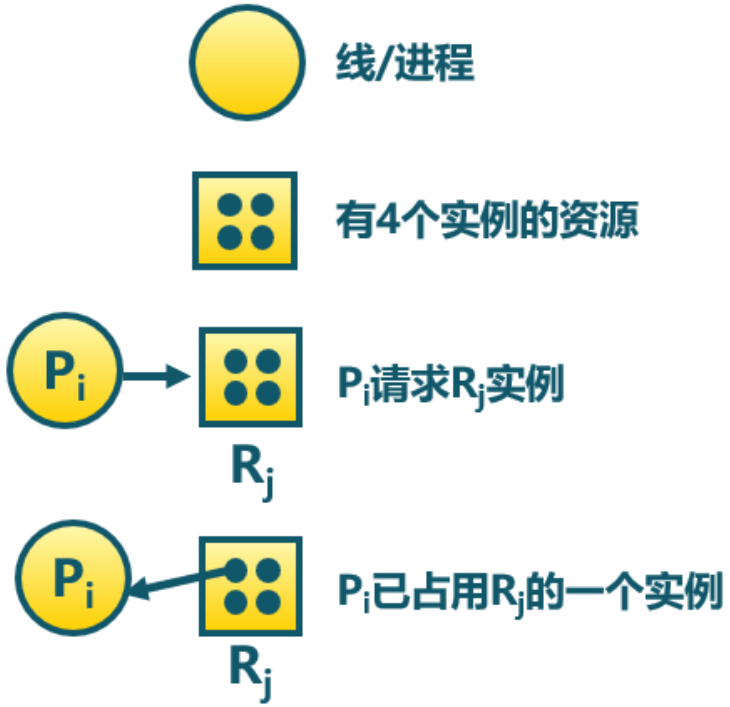
- 可消耗资源(Consumable resource)
  - 资源可被销毁
  - 可消耗资源示例
    - 在I/O缓冲区的中断、信号、消息等
  - 可能出现死锁：进程间相互等待接收对方的消息



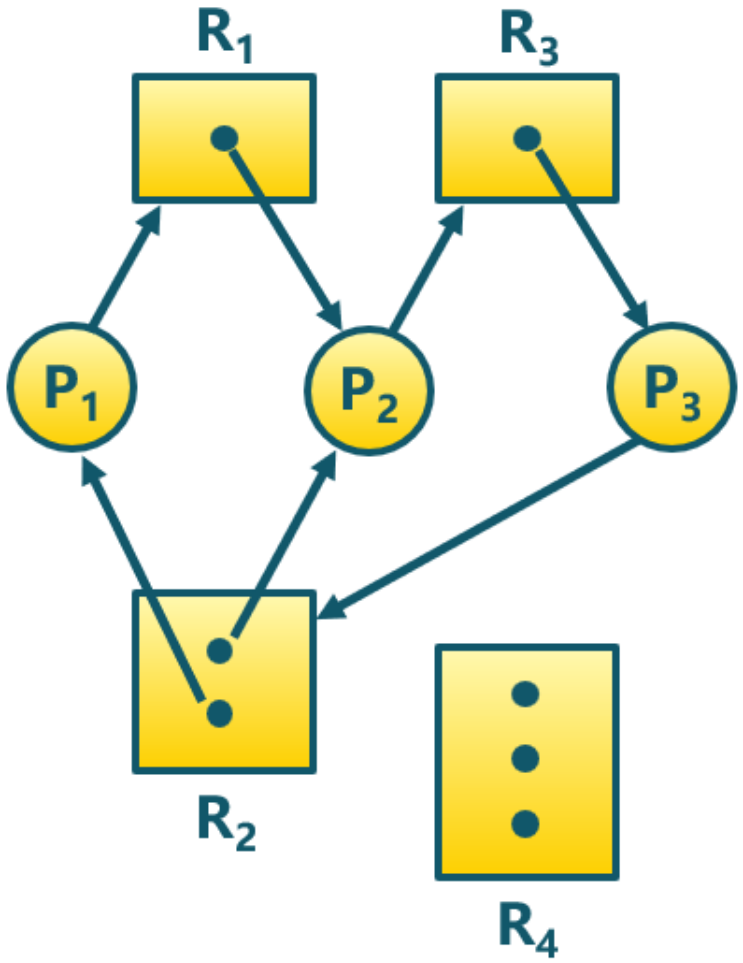
死锁问题 -- 资源分配图

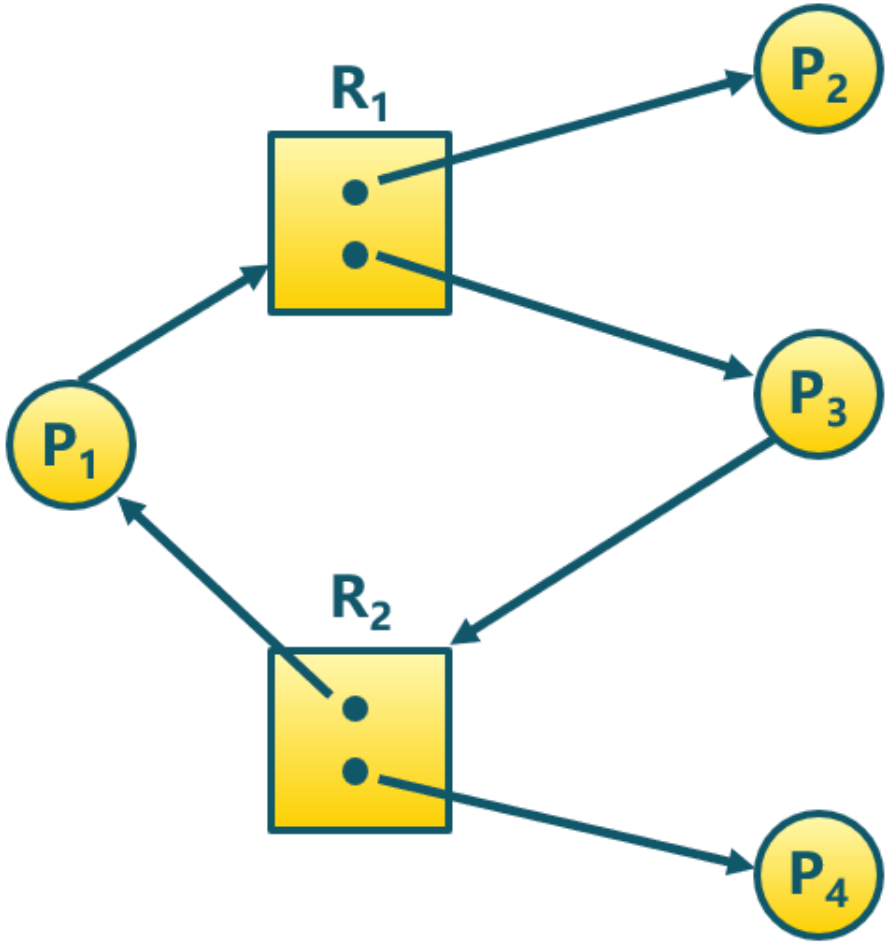
描述资源和进程间的分配和占用关系的有向图

- 顶点：系统中的进程
  - $P = \{P_1, P_2, \ldots, P_n\}$
- 顶点：系统中的资源
  - $R = \{R_1, R_2, \ldots, P_m\}$
- 边：资源请求
  - 进程P_i请求资源R_j：$P_i \rightarrow R_j$
- 边：资源分配
  - 资源R_j已分配给进程P_i：$R_j \rightarrow P_i$



死锁问题 -- 资源分配图





是否有死锁?

死锁问题 -- 必要条件

- 互斥
  - 任何时刻只能有一个进/线程使用一个资源实例
- 持有并等待
  - 进/线程保持至少一个资源，并正在等待获取其他进程持有的资源
- 非抢占
  - 资源只能在进程使用后自愿释放
- 循环等待
  - 存在等待进程集合$\{P_0, P_1, \ldots, P_N\}$
  - 进程间形成相互等待资源的环

死锁问题 -- 处理办法

- 死锁预防(Deadlock Prevention)
  - 确保系统永远不会进入死锁状态
- 死锁避免(Deadlock Avoidance)
  - 在使用前进行判断，只允许不会出现死锁的进程请求资源
- 死锁检测和恢复(Deadlock Detection & Recovery)
  - 在检测到运行系统进入死锁状态后，进行恢复
- 由应用进程处理死锁
  - 通常操作系统忽略死锁
    - 大多数操作系统（包括UNIX）的做法

死锁问题 - 处理办法 -- 预防

预防采用某种策略限制并发进程对资源的请求，或破坏死锁必要条件。

- 破坏“互斥”
  - 把互斥的共享资源封装成可同时访问，例如用SPOOLing技术将打印机改造为共享设备；
  - 缺点：但是很多时候都无法破坏互斥条件。
- 破坏“持有并等待“
  - 只在能够同时获得所有需要资源时，才执行分配操作
  - 缺点：资源利用率低

死锁问题 - 处理办法 -- 预防

预防采用某种策略限制并发进程对资源的请求，或破坏死锁必要条件。

- 破坏“非抢占”
  - 如进程请求不能立即分配的资源，则释放已占有资源
  - 申请的资源被其他进程占用时，由OS协助剥夺
  - 缺点：反复地申请和释放资源会增加系统开销，降低系统吞吐量。
- 破坏“循环等待“
  - 对资源排序，要求进程按顺序请求资源
  - 缺点：必须按规定次序申请资源，用户编程麻烦
  - 缺点：难以支持资源变化（例如新资源）

死锁问题 - 处理办法 -- 避免

利用额外的先验信息，在分配资源时判断是否会出现死锁，只在不会死锁时分配资源

- 要求进程声明需要资源的最大数目
- 限定提供与分配的资源数量，确保满足进程的最大需求
- 动态检查的资源分配状态，确保不会出现环形等待

死锁问题 - 处理办法 -- 避免

资源分配中，系统处于安全状态

- 针对所有已占用进程，存在安全执行序列$< P_1，P_2，...，P_N >$
- P_i要求的资源 $\leq$ 当前可用资源 + 所有P_j 持有资源，其中$j < i$
- 如P_i的资源请求不能立即分配，则P_i等待所有$P_j(j < i)$完成
- P_i完成后，P_{i+1}可得到所需资源，执行完并释放所分配的资源
- 最终整个序列的所有Pi都能获得所需资源

死锁问题 - 处理办法 -- 避免

安全状态与死锁的关系

- 系统处于安全状态，一定没有死锁
- 系统处于不安全状态，可能出现死锁
  - 避免死锁就是确保系统不会进入不安全状态

死锁问题 - 处理办法 -- 避免

银行家算法（Banker's Algorithm）-- 概述

- 银行家算法是一个避免死锁产生的算法。以银行借贷分配策略为基础，判断并保证系统处于安全状态
  - 客户在第一次申请贷款时，声明所需最大资金量，在满足所有贷款要求并完成项目时，及时归还
  - 在客户贷款数量不超过银行拥有的最大值时，银行家尽量满足客户需要

银行家 ↔操作系统；资金 ↔资源；客户 ↔线/进程

死锁问题 - 处理办法 -- 避免

银行家算法（Banker's Algorithm）-- 数据结构

n = 线程数量, m = 资源类型数量

- **Max（总需求量）：n×m矩阵**
  线程T_i最多请求类型R_j的资源 Max[i,j] 个实例
- **Available（剩余空闲量）： 长度为m的向量**
  当前有 Available[j] 个类型R_j的资源实例可用
- **Allocation（已分配量）： n×m矩阵**
  线程T_i 当前分配了 Allocation[i, j] 个R_j的实例
- **Need（未来需要量）： n×m矩阵**
  线程T_i 未来需要 Need[i, j] 个R_j资源实例

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

死锁问题 - 处理办法 -- 避免

银行家算法（Banker's Algorithm）-- 判断安全状态的例程

1.Work 和Finish 分别是长度为m和n的向量初始化:

Work = Available **//当前资源剩余空闲量**

Finish[i] = false for i: 1,2, ..., n. **//线程i没结束**

2.寻找线程T_i:

(a) Finish[i] = false **//接下来找出Need比Work小的线程i**

(b) Need[i]≤Work

没有找到满足条件的T_i，转4。

3.Work = Work + Allocation[i] **//线程i的资源需求量小于当前剩余空闲资源量, 所以配置给它再回收**

Finish[i] = true

转2.

4.如所有线程Ti满足Finish[i] == true，则系统处于安全状态 **//所有线程的Finish为True, 表明系统处于安全状态**

死锁问题 - 处理办法 -- 避免

银行家算法（Banker's Algorithm）-- 完整算法

初始化: $Request_i$ 线程T_i的资源请求向量
$Request_i[j]$ 线程T_i请求资源R_j的实例

循环:

1.如果 $Request_i \le Need[i]$, 转到步骤2。否则, 拒绝资源申请,
因为线程已经超过了其最大要求

2.如果 $Request_i \le Available$, 转到步骤3。否则, T_i 必须等待,
因为资源不可用

3.通过安全状态判断来确定是否分配资源给T_i :
生成一个需要判断状态是否安全的资源分配环境

$Available = Available - Request_i;$
$Allocation[i] = Allocation[i] + Request_i;$
$Need[i] = Need[i] - Request_i;$

调用判断安全状态的例程

如果返回结果是安全，将资源分配给T_i
如果返回结果是不安全，系统会拒绝T_i的资源请求

死锁问题 - 处理办法 -- 避免

银行家算法（Banker's Algorithm）-- 示例1

初始状态

	R1	R2	R3
T1	3	2	2
T2	6	1	3
T3	3	1	4
T4	4	2	2

最大需求矩阵 C

	R1	R2	R3
T1	1	0	0
T2	6	1	2
T3	2	1	1
T4	0	0	2

已分配资源矩阵 A

	R1	R2	R3
T1	2	2	2
T2	0	0	1
T3	1	0	3
T4	4	2	0

当前资源请求矩阵C-A

R1	R2	R3
9	3	6

系统资源向量R

R1	R2	R3
0	1	1

当前可用资源向量V

死锁问题 - 处理办法 -- 避免

银行家算法（Banker's Algorithm）-- 示例1

线程T2完成运行

	R1	R2	R3
T1	3	2	2
T2	0	0	0
T3	3	1	4
T4	4	2	2

最大需求矩阵 C

	R1	R2	R3
T1	1	0	0
T2	0	0	0
T3	2	1	1
T4	0	0	2

已分配资源矩阵 A

	R1	R2	R3
T1	2	2	2
T2	0	0	0
T3	1	0	3
T4	4	2	0

当前资源请求矩阵C-A

R1	R2	R3
9	3	6

系统资源向量R

R1	R2	R3
6	2	3

当前可用资源向量V

死锁问题 - 处理办法 -- 避免

银行家算法（Banker's Algorithm）-- 示例1

死锁问题 - 处理办法 -- 避免

银行家算法（Banker's Algorithm）-- 示例1

线程T3完成运行

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	0	0	0
T4	4	2	2

最大需求矩阵 C

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	0	0	0
T4	0	0	2

已分配资源矩阵 A

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	0	0	0
T4	4	2	0

当前资源请求矩阵C-A

R1	R2	R3
9	3	6

系统资源向量R

R1	R2	R3
9	3	4

当前可用资源向量V

死锁问题 - 处理办法 -- 避免

银行家算法（Banker's Algorithm）-- 示例2

初始状态

	R1	R2	R3
T1	3	2	2
T2	6	1	3
T3	3	1	4
T4	4	2	2

最大需求矩阵 C

	R1	R2	R3
T1	1	0	0
T2	5	1	1
T3	2	1	1
T4	0	0	2

已分配资源矩阵 A

	R1	R2	R3
T1	2	2	2
T2	1	0	2
T3	1	0	3
T4	4	2	0

当前资源请求矩阵C-A

R1	R2	R3
9	3	6

系统资源向量R

R1	R2	R3
1	1	2

当前可用资源向量V

死锁问题 - 处理办法 -- 避免

银行家算法（Banker's Algorithm）-- 示例2

线程T1请求R1和R3资源各1个实例

	R1	R2	R3
T1	3	2	2
T2	6	1	3
T3	3	1	4
T4	4	2	2

最大需求矩阵 C

	R1	R2	R3
T1	2	0	1
T2	5	1	1
T3	2	1	1
T4	0	0	2

已分配资源矩阵 A

	R1	R2	R3
T1	1	2	1
T2	1	0	2
T3	1	0	3
T4	4	2	0

当前资源请求矩阵C-A

R1	R2	R3
9	3	6

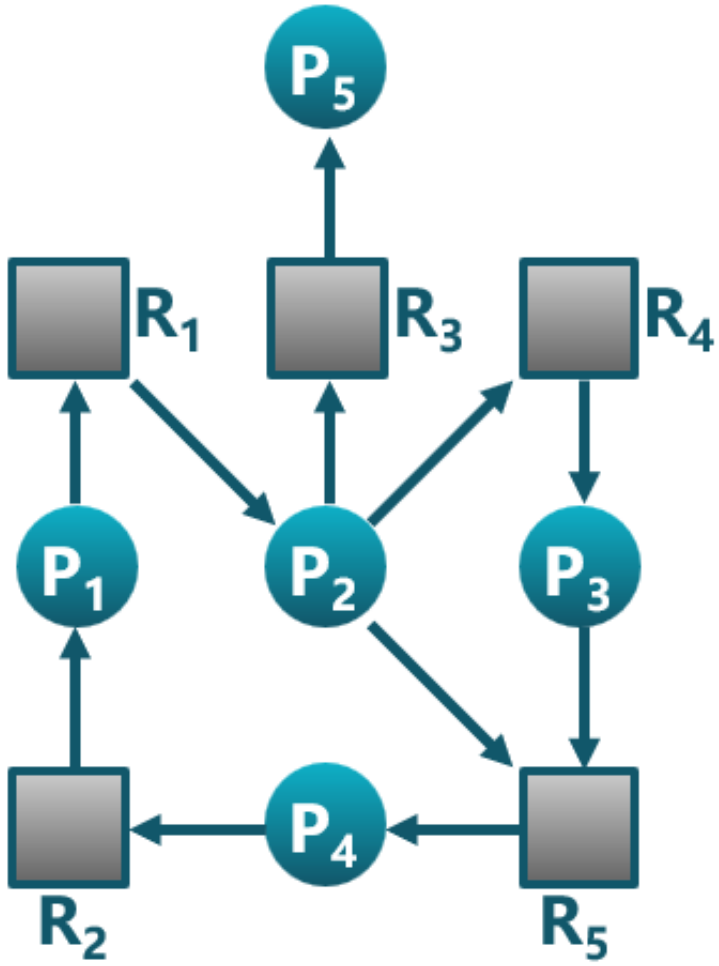
系统资源向量R

R1	R2	R3
0	1	1

当前可用资源向量V

死锁问题 - 处理办法 -- 检测

- 允许系统进入死锁状态
- 维护系统的资源分配图
- 定期调用死锁检测算法来搜索图中是否存在死锁
- 出现死锁时，用死锁恢复机制进行恢复



死锁问题 - 处理办法 -- 检测

死锁检测算法：数据结构

- Available:长度为m的向量：每种类型可用资源的数量
- Allocation:一个n×m矩阵：当前分配给各个进程每种类型资源的数量
  - 进程P_i 拥有资源R_j的$Allocation[i,j]$个实例

死锁问题 - 处理办法 -- 检测

死锁检测算法：完整算法

1.Work 和Finish 分别是长度为m和n的向量初始化:

(a)Work = Available　　//work为当空闲资源量

(b)Allocation[i]> 0 时, Finish[i] = false;　　//finish为线程是否结束

否则，Finish[i] = true

2.寻找线程T_i满足:

(a)Finish[i] = false　　//线程没有结束的线程，且此线程将需要的资源量小于当前空闲资源量

(b)$Request_i \leq Work$

没有找到这样的i，转到4

3.Work = Work + Allocation[i]　　//把找到的线程拥有的资源释放回当前空闲资源中

Finish[i] = true

转到2

4.如某个Finish[i] == false，系统处于死锁状态　　//如果有Finish为false,表明系统处于死锁状态

死锁问题 - 处理办法 -- 检测

死锁检测算法：-- 示例1

- 5个线程T_0 到 T_4；3种资源类型
  A (7个实例), B (2个实例), and C (6个实例)
- 在T_0时刻：

	已分配资源			资源请求			当前可用资源		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	0	0
T_1	2	0	0	2	0	2			
T_2	3	0	3	0	0	0			
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

死锁问题 - 处理办法 -- 检测

死锁检测算法：-- 示例1

- 5个线程T_0 到 T_4；3种资源类型
  A (7个实例), B (2个实例), and C (6个实例)
- 在T_0时刻：

	已分配资源			资源请求			当前可用资源		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	1	0
T_1	2	0	0	2	0	2			
T_2	3	0	3	0	0	0			
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

死锁问题 - 处理办法 -- 检测

死锁检测算法：-- 示例1

- 5个线程T_0 到 T_4；3种资源类型
  A (7个实例), B (2个实例), and C (6个实例)
- 在T_0时刻：

	已分配资源			资源请求			当前可用资源		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	1	0
T_1	2	0	0	2	0	2			
T_2	3	0	3	0	0	0	3	1	3
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

死锁问题 - 处理办法 -- 检测

死锁检测算法：-- 示例1

- 5个线程T_0 到 T_4；3种资源类型
  A (7个实例), B (2个实例), and C (6个实例)
- 在T_0时刻：

	已分配资源			资源请求			当前可用资源		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	1	0
T_1	2	0	0	2	0	2	5	1	3
T_2	3	0	3	0	0	0	3	1	3
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

死锁问题 - 处理办法 -- 检测

死锁检测算法：-- 示例1

- 5个线程T_0 到 T_4；3种资源类型
  A (7个实例), B (2个实例), and C (6个实例)
- 在T_0时刻：

	已分配资源			资源请求			当前可用资源		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	1	0
T_1	2	0	0	2	0	2	5	1	3
T_2	3	0	3	0	0	0	3	1	3
T_3	2	1	1	1	0	0	7	2	4
T_4	0	0	2	0	0	2			

死锁问题 - 处理办法 -- 检测

死锁检测算法：-- 示例1

- 5个线程T_0 到 T_4；3种资源类型
  A (7个实例), B (2个实例), and C (6个实例)
- 在T_0时刻：

	已分配资源			资源请求			当前可用资源		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	1	0
T_1	2	0	0	2	0	2	5	1	3
T_2	3	0	3	0	0	0	3	1	3
T_3	2	1	1	1	0	0	7	2	4
T_4	0	0	2	0	0	2	7	2	6

序列$< T_0, T_2, T_1, T_3, T_4 >$对于所有的i，都可满足Finish[i] = true

死锁问题 - 处理办法 -- 检测

死锁检测算法：-- 示例2

- 5个线程T_0 到 T_4；3种资源类型
  A (7个实例), B (2个实例), and C (6个实例)
- 在T_0时刻：

	已分配资源			资源请求			当前可用资源		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	0	0
T_1	2	0	0	2	0	1			
T_2	3	0	3	0	0	1			
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

可通过回收线程T_0占用的资源，但资源不足以完成其他线程请求

线程T_1, T_2, T_3, T_4形成死锁

死锁问题 - 处理办法 -- 检测

使用死锁检测算法

- 死锁检测的时间和周期选择依据
  - 死锁多久可能会发生
  - 多少进/线程需要被回滚
- 资源图可能有多个循环
  - 难于分辨“造成”死锁的关键进/线程

检测到死锁后，应该如何处理?

死锁问题 - 处理办法 -- 恢复 -- 进程终止

- 终止所有的死锁进程
- 一次只终止一个进程直到死锁消除
- 终止进程的顺序的参考因素:
  - 进程的优先级
  - 进程已运行时间以及还需运行时间
  - 进程已占用资源
  - 进程完成需要的资源
  - 终止进程数目
  - 进程是交互还是批处理

死锁问题 - 处理办法 -- 恢复 -- 资源抢占

- 选择被抢占进程
  - 参考因素：最小成本目标
- 进程回退
  - 返回到一些安全状态, 重启进程到安全状态
- 可能出现饥饿
  - 同一进程可能一直被选作被抢占者