死锁避免

死锁避免 (Deadlock Avoidance)

- 前提：系统拥有先验知识 (a priori information)，知道每个进程将如何利用资源
- 简单而直观（但是不准确）的要求，则是每个进程事先申报：每种类型资源的最大需求数

死锁避免（续）

- 死锁避免算法动态地检测资源分配状态(resource-allocation state)，它总是确保 circular-wait 条件永远不成立
- 资源分配状态决定于 3 个因素
- 可分配资源数
- 已分配资源数
- 进程对每种类型资源的最大需求数

安全状态 (Safe State)

- 也即安全的资源分配状态
- 当一进程申请可用资源时，系统必须判断：
- 此次分配之后，系统是否仍然处于“安全状态”
- 如果存在一个序列 <P_1, P_2, …, P_n>
- 进程 P_i 将申请的资源总量 <
- （当前可用资源 + 所有进程 P_j 占有的资源）
- $j < i$
- 那么，称该系统处于“安全状态”

安全状态（续）

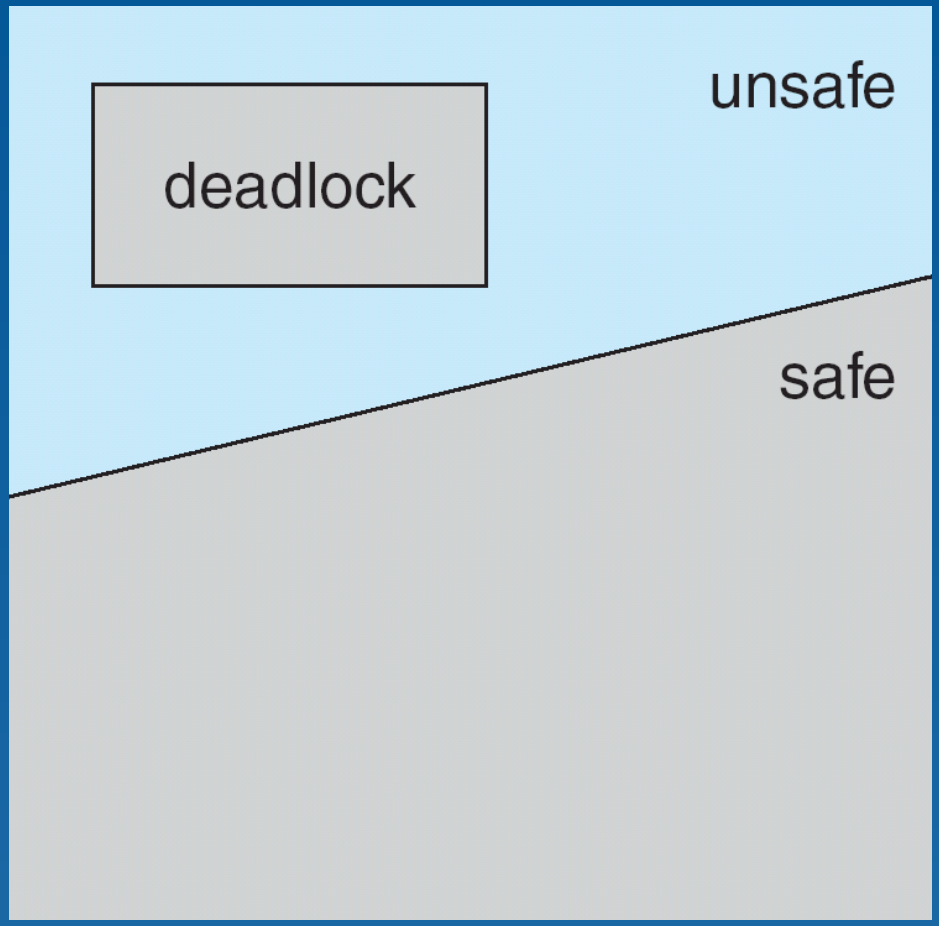
- 对定义的解读
  - 如果进程 P_i 需要的资源不能立即到位，它可以等待有限时间，直至进程 P_j 释放相关资源
  - 进程 P_j 运行结束时，进程 P_i 即从进程 P_j 获得它需要的资源。然后执行，然后执行完毕，然后释放所有资源
  - 进程 P_i 结束时，进程 P_{i+1} 从进程 P_i 获得它需要的资源。与此类推

安全状态（续）

- 对定义的解读
- 如果系统总是处于安全状态 ⇒ 不会死锁
- 如果系统某时刻不在安全状态 ⇒ 可能死锁，不是一定死锁

确保系统用不进入非“安全状态” ⇒ 死锁避免

安全，非安全，死锁



Dijkstra 的银行家算法 (Banker's Algorithm)

- 算法能够管理多类型、多实例的资源
- 每个进程必须事先预报它将用到的最大资源数
- 一个进程申请资源后，它可能被要求等待
- 一个进程得到所有资源后，必须在有限时间内归还

银行家算法的数据结构

令 n = 进程总数， m = 资源类型总数

- ***Available*** - 长度为 m 的矢量

available [j] = k 表示 R_j 类型的资源共有 k 个实例可用

- ***Max*** - n x m 矩阵

Max [i,j] = k 表示进程 P_i 可申请最多 k 个 R_j 类型的资源

银行家算法的数据结构

- ***Allocation*** - *n* x *m* 矩阵

Allocation[*i,j*] = *k* 表示进程 P_i 拥有 R_j 类型的资源 *k* 个

- ***Need*** - *n* x *m* 矩阵

Need[*i,j*] = *k* 表示进程 P_i 还需要申请 R_j 类型的资源 *k* 个，才能完成任务

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j]$$

安全状态判断算法 Safety

1. 令 ***Work*** 为长度 *m* 的矢量， ***Finish*** 为长度 *n* 的矢量。初始值:

   Work = *Available*

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

2. 选取满足如下条件的 *i*

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

   (b) $Need_i \leq Work$

   If 不存在这样的 *i* ， then 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 系统处于安全状态，返回 safe

银行家算法响应进程 P_i 的请求

定义 *Request* 矢量。 $Request_i[j] = k$ 表示进程 P_i 发出申请，申请 R_j 类型的资源 k 个

1. If $Request_i \le Need_i$ go to step 2.

否则，报错。出错原因是进程申请数超过了预报数

2. If $Request_i \le Available$, go to step 3.

否则，进程 P_i 等待，因为资源不够用

3. 假如，如进程 P_i 所愿，它得到了申请的资源。那么：

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

4. 调用 Safety 算法

- If 返回 *safe* ⇒ 不会死锁，资源如数分配给进程 *Pi.*
- If 返回 *unsafe* ⇒ 进程 *Pi* 等待，数据结构 Available 、 $Allocation_i$ 、 $Need_i$ 恢复至调用算法前的值

银行家算法示例

- 5 个进程 P_0 至 P_4; 3 类资源:

  A (10 个实例), B (5 个实例), 以及 C (7 个实例)

- 在 T_0 时刻有

	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	

银行家算法示例（续）

- 矩阵 *Need* 就是 *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

- Safety 算法得到了安全序列 $< P_1, P_3, P_4, P_2, P_0 >$，因此系统处于安全状态

这时，进程 P_1 申请 Request (1,0,2)

◆由计算知 Request ≤ Available (i.e., (1,0,2) ≤ (3,3,2))

	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 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

◆Safety 算法得到了安全序列 < P_1, P_3, P_4, P_0, P_2> ， 因此系统处于安全状态。银行家算法允许资源分配

◆这时，进程 P_4 申请 request (3,3,0) ， be granted?


End