



3.6 死锁 Deadlock

死锁产生

死锁防止

死锁避免

死锁的检测和解除





死锁产生

死锁防止

死锁避免

死锁的检测和解除



3.6.1 死锁产生

- 独占性资源, 如磁带机、打印机、绘图仪等硬件设备以及进程表、临界区等软件资源不能同时供多个进程使用, 否则容易导致结果混乱、数据错误以及程序崩溃, 因此系统一次仅允许一个进程访问独占性资源
- 如果多个进程共享的资源为独占性资源, 处理不当, 就可能发生若无外力, **进程永远相互等待**的情况, 这时就说这组进程发生了死锁



河海大学

计算机与信息学院

时间片	进程P	进程Q
1	请求读卡机	
2		请求打印机
3	请求打印机	
4		请求读卡机
5	释放读卡机	
6		释放读卡机
7	释放打印机	
8		释放打印机



3.6.1: 例

进程推进顺序不当产生死锁

设系统有打印机、读卡机各一台，被进程P和Q共享。两个进程并发执行，按下列次序请求和释放资源：

时间片	进程P	进程Q
1	请求读卡机	
2		请求打印机
3	请求打印机	
4		请求读卡机
5	释放读卡机	
6		释放读卡机
7	释放打印机	
8		释放打印机



3.6.1: 例 (2)

PV操作使用不当产生死锁

$S1=1, S2=1$

时间片	进程Q1	进程Q2
1
2	P(S1)	
3		P(S2)
4	P(S2)	
5		P(S1)
6
7	V(S1)	
8		V(S2)
9	V(S2)	
10		V(S1)



3.6.1: 例 (3)

资源分配不当引起死锁

若系统中有 m 个资源被 n 个进程共享，每个进程都要求 K 个资源，而 $m < n \cdot K$ 时，即资源数小于进程所要求的总数时，如果分配不得当就可能引起死锁

例如三个进程 $P1$ 、 $P2$ 、 $P3$ 都需要3个同类资源，系统共有该种资源4个，则分配不当时就会产生死锁如：

时间片	P1	P2	P3
1	1		
2		1	
3			1
4	3		
5		?	
6			?

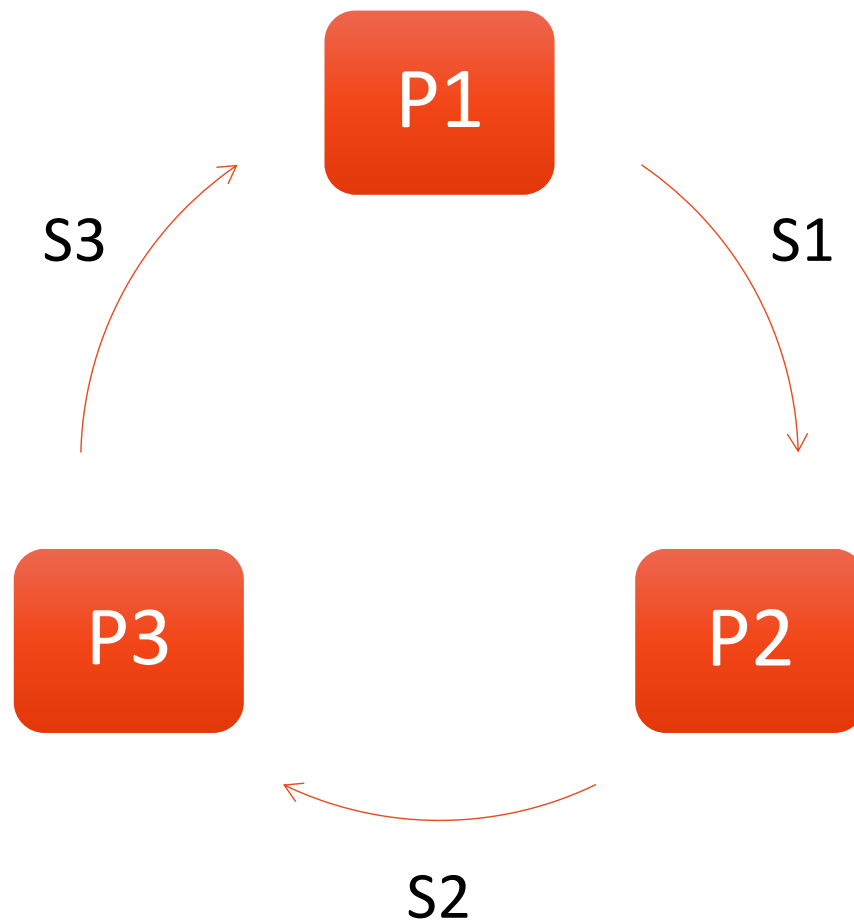


3.6.1: 例 (4)

对临时性资源使用不加限制

引起死锁

进程通信使用的信件是一种临时性资源, 如果对信件的发送和接收不加限制, 可能引起死锁





3.6.1 (续)

- 在讨论死锁问题时，特作如下假定：
 - **假定1：**任意一个进程要求资源的最大数量不超过系统能提供的最大量
 - **假定2：**如果一个进程在执行中提出的资源要求能够得到满足，那么，它一定能在有限的时间内结束
 - **假定3：**一个资源在任何时间最多只为一个进程所占有
 - **假定4：**一个进程一次申请一个资源，且只在申请资源得不到满足时才处于等待状态
 - **假定5：**一个进程结束时释放它所占有的全部资源
 - **假定6：**系统具有有限个进程和有限个资源



3.6.1：死锁定义

- 死锁的定义

- 操作系统中的死锁指：如果在一个进程集合中的每个进程都在等待只能由该集合中的其他一个进程才能引发的事件，则称一组进程或系统此时发生了死锁

- 另一种定义：

- 所谓死锁，是指多个进程在运行过程中因争夺资源而造成的一种僵局，当进程处于这种僵局状态时，若无外力作用，他们都将无法再向前推进



3.6.1：死锁产生因素

- 与系统拥有的资源数量有关
- 与资源分配策略有关
- 与进程对资源的使用要求以及并发进程的推进顺序有关



3.6.1：死锁产生因素

- 与系统拥有的资源数量有关
- 与资源分配策略有关
- 与进程对资源的使用要求以及并发进程的推进顺序有关

处理死锁的基本方法

- 预防死锁 (3.6.2)
- 避免死锁 (3.6.3)
- 检测死锁 (3.6.4)
- 解除死锁 (3.6.4)



死锁产生

死锁防止

死锁避免

死锁的检测和解除



3.6.2: 产生死锁的必要条件



Coffman E G,
Elphick M,
Shoshani A.

System deadlocks[J].
ACM Computing
Surveys (CSUR),
1971, 3(2): 67-78.

- **互斥条件**: 进程互斥使用资源
- **占有和等待条件 (部分分配条件)**: 申请新资源时不释放已占有资源
- **不剥夺条件**: 一个进程不能抢夺其他进程占有的资源
- **循环等待条件 (环路条件)**: 存在一组进程循环等待资源的现象



3.6.2: 产生死锁的必要条件

是死锁产生的
必要条件, 不
是充分条件

- **互斥条件**: 进程互斥使用资源
- **占有和等待条件 (部分分配条件)**: 申请新资源时不释放已占有资源
- **不剥夺条件**: 一个进程不能抢夺其他进程占有的资源

是前三个条件
同时存在时产
生的结果

- **循环等待条件 (环路条件)**: 存在一组进程循环等待资源的现象



3.6.2: 产生死锁的必要条件

只要破坏
这四个条
件之一，
死锁就可
防止！！

- 互斥条件：进程互斥使用资源
- 占有和等待条件(部分分配条件)：申请新资源时不释放已占有资源
- 不剥夺条件：一个进程不能抢夺其他进程占有的资源
- 循环等待条件(环路条件)：存在一组进程循环等待资源的现象



3.6.2: 产生死锁的必要条件

- **互斥条件**: 进程互斥使用资源
- **占有和等待条件 (部分分配条件)**: 申请新资源时不释放已占有资源
- **不剥夺条件**: 一个进程不能抢夺其他进程占有的资源
- **循环等待条件 (环路条件)**: 存在一组进程循环等待资源的现象

使资源可同时访问而不是互斥使用的办法**对于磁盘适用**, 但对于磁带机、打印机等多数资源来说不仅不能破坏互斥使用条件, 还要加以保证



3.6.2: 产生死锁的必要条件


- **互斥条件**: 进程互斥使用资源
- **占有和等待条件 (部分分配条件)**: 申请新资源时不释放已占有资源
- **不剥夺条件**: 一个进程不能抢夺其他进程占有的资源
- **循环等待条件 (环路条件)**: 存在一组进程循环等待资源的现象

采用剥夺式调度方法可以破坏第三个条件（不剥夺条件），但剥夺调度方法目前只适用于对主存资源和处理器资源的分配



3.6.2: 产生死锁的必要条件

- **互斥条件**: 进程互斥使用资源
- **占有和等待条件(部分分配条件)**: 申请新资源时不释放已占有资源
- **不剥夺条件**: 一个进程不能抢夺其他进程占有的资源
- **循环等待条件(环路条件)**: 存在一组进程循环等待资源的现象



比较实用的死锁防止方法是破坏第二个条件（占有和等待条件）的静态分配策略和破坏第四个条件（循环等待条件）的层次分配策略



3.6.2: 静态分配策略

静态分配是指一个进程必须在执行前就申请它所要的全部资源，并且直到它所要的资源都得到满足后才开始执行

降低了资源利用率, 因为每个进程占有的资源中, 有些资源在较后的时间里才使用, 有些资源在发生例外时才使用, 这样就可能造成一个进程占有了一些几乎不用的资源而使其它想用这些资源的进程产生等待



3.6.2: 层次分配策略

- 资源被分成多个层次
- 当进程得到某一层的一个资源后，它只能再申请较高层次的资源
- 当进程要释放某层的一个资源时，必须先释放占有的较高层次的资源
- 当进程得到某一层的一个资源后，它想申请该层的另一个资源时，必须先释放该层中的已占资源



3.6.2: 层次分配策略

P_A

1. P(S1);
2. P(S2);
3. ...
4. V(S2);
5. V(S1);

P_B

1. P(S1);
2. P(S3);
3. ...
4. V(S3);
5. V(S1);

P_C

1. P(S1);
 2. P(S2);
 3. P(S3);
 4. ...
 5. V(S3);
 6. V(S2);
 7. V(S1);
-

层次策略的变种: 按序分配策略



3. 6. 2: 层次分配策略（例）

- 系统中有四类资源编号为：1. 输入机 4. 打印机 7. 磁带机 9. 磁盘机
- 现有两个进程P1、P2都要使用打印机和磁盘机， P1先使用打印机后使用磁盘机， P2先使用磁盘机后使用打印机。如果P2先提出资源请求，则P2只能先申请打印机（编号小），然后申请磁盘机（编号大）。 P2在使用磁盘机的时候，打印机空闲着不能被P1申请使用



3.6.2: 层次分配策略（缺点）

1. 限制了新类型设备的增加
2. 进程使用各类资源顺序与系统规定顺序不同时造成资源浪费
3. 限制用户简单、自主编程

死锁的预防条件很苛刻，导致系统运行效率低下。可以允许所有3个必要条件的存在，但避免第四个条件产生即可，即**死锁避免：银行家算法**（3.6.3）



死锁产生

死锁防止

死锁避免

死锁的检测和解除



3.6.3 死锁的避免

- 银行家算法是由Dijkstra提出的, 算法思想如下:
- 一个银行家 (操作系统) 拥有资金M (可用资源), 被N个客户 (进程) 共享, 银行家对客户提出下列约束条件:
 - 每个客户必须预先说明自己所要求的最大资金量
 - 每个客户每次提出部分资金量申请和获得分配
 - 如果银行满足了客户对资金的最大需求量, 则客户在资金运作后一定可以很快归还资金



3.6.3: 银行家算法基本思想

1. 系统中的所有进程进入进程集合,
2. 在安全状态下系统收到进程的资源请求后, 先把资源**试探性分配**给它
3. 系统用剩下的可用资源和进程集合中其他进程还要的资源数作比较, 在进程集合中**找到剩余资源能满足最大需求量的进程**, 从而保证这个进程运行完毕并归还全部资源
4. 把这个进程从集合中去掉, 系统的剩余资源更多了, 反复执行上述步骤
5. 最后, 检查进程集合, 若为空表明本次申请可行, 系统处于安全状态, 可实施本次分配; 否则, 有进程执行不完, 系统处于不安全状态, 本次资源分配暂不实施, 让申请进程等待



3.6.3: 例

- 有三个银行客户P1, P2, P3需要向某银行分别借款10万元、4万元、9万元, 该银行共有12万元资金可供贷出。在T0时刻, 该银行分别向三个客户提供贷款5万元、2万元、2万元, 银行尚有资金3万元。

客户	最大需求	已分配	可用
P1	10	5	3
P2	4	2	
P3	9	2	



3.6.3: 例 (续)

客户	C	$A(T_0)$	T_1	T_2	T_3	T_4	T_5	T_6
P1	10	5	5	5	10	0	0	0
P2	4	2	4	0	0	0	0	0
P3	9	2	2	2	2	2	9	0
V	12	3	1	5	0	10	3	12



3.6.3: 例 (续)

客户	C	A(T ₀)	T ₁	T ₂	T ₃	T ₄	T ₅	T ₆
P1	10	5	5	5	10	0	0	0
P2	4	2	4	0	0	0	0	0
P3	9	2	2	2	2	2	9	0

- **安全状态**: 指系统能按某种进程顺序 (P1, P2, , Pn) (称 $\langle P1, P2, \dots, Pn \rangle$ 序列为**安全序列**), 来为每个进程Pi分配其所需资源, 直至满足每个进程对资源的最大需求, 使每个进程都能顺利完成 {P2, P1, P3}
- 如果找不到这样的安全序列, 就称系统处于不安全状态



3.6.3 (续)

- 安全状态、不安全状态与死锁状态的关系
 - 并非所有不安全状态都是死锁状态，当系统进入不安全状态后，便可能进入死锁状态
 - 只要系统处于安全状态，便可避免进入死锁状态
 - 避免死锁的实质：系统在分配资源时，保证系统不进入不安全状态



3.6.3: 例（不安全状态）

- 实例中, 假如在 T_0 时刻的基础上, 客户P3请求贷款1万元, 银行满足了这个请求

客户	C	$A(T_0)$	T_1	T_2	T_3	T_4
P1	10	5	5	5	5	不安全
P2	4	2	2	4	0	
P3	9	2	3	3	3	
V	12	3	2	0	4	



3.6.3: 银行家算法

```
typedef struct state
```

```
int resource[m];          //R
```

```
int request[m];
```

```
int available[m];        //V
```

```
int claim[n][m];         //C
```

```
int allocation[n][m];    //A
```

$$R = V + \sum_{k=1}^n A_k$$

$$C_k \leq R \quad (k = 1, \dots, n)$$

$$A_k \leq C_k \quad (k = 1, \dots, n)$$

若要启动一个新进程，在最坏情况下，则其需求 $C_{(n+1)}$ 应该满足：

$$R \geq C_{(n+1)} + \sum_{k=1}^n C_k$$

进程启动拒绝法



3.6.3: 银行家算法

```
typedef struct state
```

```
int resource[m];          //R
```

```
int request[m];
```

```
int available[m];        //V
```

```
int claim[n][m];         //C
```

```
int allocation[n][m];    //A
```

$$R = V + \sum_{k=1}^n A_k$$

$$C_k \leq R \quad (k = 1, \dots, n)$$

$$A_k \leq C_k \quad (k = 1, \dots, n)$$

存在进程序列 $\{P_1, P_2, \dots, P_n\}$, 对于 P_k ($1 \leq k \leq n$), 满足:

$$C_k - A_k \leq V + \sum_{j=1}^k A_j$$

安全状态: 若进程 P_k 的需求不能被立即满足, 那么当其左边序列内的所有进程结束后, 即可满足。



3.6.3: 银行家算法

```
void resource_allocation()
```

```
1.  if (allocation[i,*] + request[*] > claim[i,*])
2.      {error}; //申请量超过最大需求值
3.  else {
4.      if (request[*] > available[*]) {suspend};
5.      else{
6.          allocation[i,*] += request[*];
7.          available[*] -= request[*];
8.      } //get newstate
9.  if (safe(newstate)) { carry out allocation};
10. else {
11.     restore original state;
12.     suspend;
13. }
14. }
```



3.6.3: 银行家算法

`bool safe(state s)`

```
1.  int currentavail[m] = available[*];
2.  set<process> rest = {all processes};
3.  bool possible = true;
4.  while(possible) { //在rest中找到Pk, 满足
5.      found = if (
6.          claim[k,*] - allocation[k,*] <= currentavail[*]);
7.      if(found) {
8.          currentavail[*] += allocation[k, *];
9.          rest -= {Pk};
10.     } else { possible = false;}
11. }
12. return (rest == null);
```



3. 6. 3: (例2)

- 如果系统中共有五个进程和A、B、C三类资源；
- A类资源共有10个, B类资源共有5个, C类资源共有7个
- 在T0 时刻:

进程	allocation			claim			C - A			available		
	A	B	C	A	B	C	A	B	C	A	B	C
P ₀	0	1	0	7	5	3	7	4	3	3	3	2
P ₁	2	0	0	3	2	2	1	2	2			
P ₂	3	0	2	9	0	2	6	0	0			
P ₃	2	1	1	2	2	2	0	1	1			
P ₄	0	0	2	4	3	3	4	3	1			



3.6.3: (例2)

- 在T0 时刻, 存在安全队列: $\{P_1, P_3, P_4, P_2, P_0\}$

	currentavail			C - A			allocation			currentavail +allocation			possible
	A	B	C	A	B	C	A	B	C	A	B	C	
P ₁	3	3	2	1	2	2	2	0	0	5	3	2	true
P ₃	5	3	2	0	1	1	2	1	1	7	4	3	true
P ₄	7	4	3	4	3	1	0	0	2	7	4	5	true
P ₂	7	4	5	6	0	0	3	0	2	10	4	7	true
P ₀	10	4	7	7	4	3	0	1	0	10	5	7	true



3. 6. 3: (例2)

- 现 P_1 需要申请1个A类资源和2个C类资源，尝试分配，得到如下新状态 S_{new} :

进程	allocation			claim			C - A			available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3	2	3	0
P_1	3	0	2	3	2	2	0	2	0			
P_2	3	0	2	9	0	2	6	0	0			
P_3	2	1	1	2	2	2	0	1	1			
P_4	0	0	2	4	3	3	4	3	1			



3.6.3: (例2)

- 需要判断新状态 S_{new} 是否安全

	currentavail			C - A			allocation			currentavail +allocation			possible
	A	B	C	A	B	C	A	B	C	A	B	C	
P ₁	2	3	0	0	2	0	3	0	2	5	3	2	true
P ₃	5	3	2	0	1	1	2	1	1	7	4	3	true
P ₄	7	4	3	4	3	1	0	0	2	7	4	5	true
P ₀	7	4	5	7	4	3	0	1	0	7	5	5	true
P ₂	7	5	5	6	0	0	3	0	2	10	5	7	true

可以找到安全序列: {P₁, P₃, P₄, P₀, P₂}, S_{new} 安全, 因此可以执行对P₁的资源分配



3.6.3: 算法缺陷

- 很难在进程运行前知道其所需的资源最大量
- 系统中的进程必须是无关系的, 相互间没有同步要求
- 进程的个数和分配的资源数目应该是固定的

这些要求事先难以满足,

因而银行家算法**缺乏实用价值**



死锁产生

死锁防止

死锁避免

死锁的检测和解除



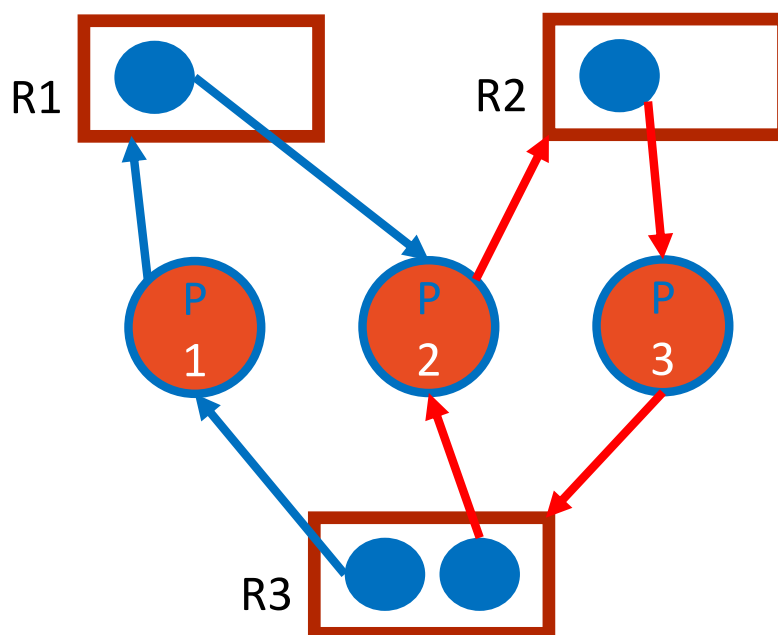
3.6.4 死锁的检测和解除

- 解决死锁问题的一条途径是死锁检测和解除，这种方法对资源的分配不加任何限制，也不采取死锁避免措施，但系统定时地运行一个“死锁检测”程序，判断系统内是否已出现死锁，如果检测到系统已发生了死锁，再采取措施解除它



3.6.4 (续)

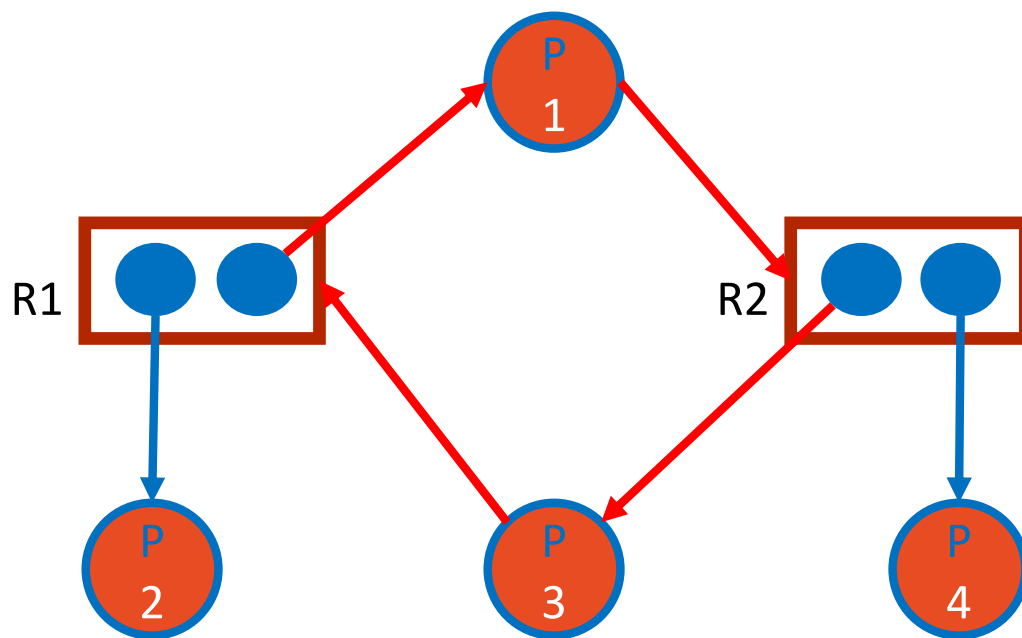
- 操作系统中每一时刻的系统状态都可以用进程—资源分配图来表示，进程—资源分配图是描述进程和资源间申请和分配关系的一种有向图，可用于检测系统是否处于死锁状态





3.6.4 (续)

- 存在环路并不一定必然发生死锁状态





3.6.4: 死锁关系判断

1. 如果进程—资源分配图中无环路，则此时系统没有发生死锁
2. 如果进程—资源分配图中有环路，且每个资源类中仅有一个资源，则系统中发生了死锁，此时，环路是系统发生死锁的充要条件，环路中的进程便为死锁进程
3. 如果进程—资源分配图中有环路，且涉及的资源类中有多个资源，则环的存在只是产生死锁的必要条件而不是充分条件



3.6.4: 简化检测方法

1. 从进程—资源分配图中找到一个既不阻塞又非独立的进程, 消去所有与该进程相连的有向边, 相当于该进程能够执行完成释放资源, 回收资源使之成为孤立结点. 然后将所回收的资源分配给其它进程,
2. 再从进程—资源分配图中找到下一个既不阻塞又非独立的进程, 消去所有与该进程相连的有向边, 使之成为孤立结点…….
3. 不断重复该过程, 直到所有进程成为孤立结点, 则称该图是**可完全简化的**; 否则称该图是**不可完全简化的**



3.6.4: 简化检测方法

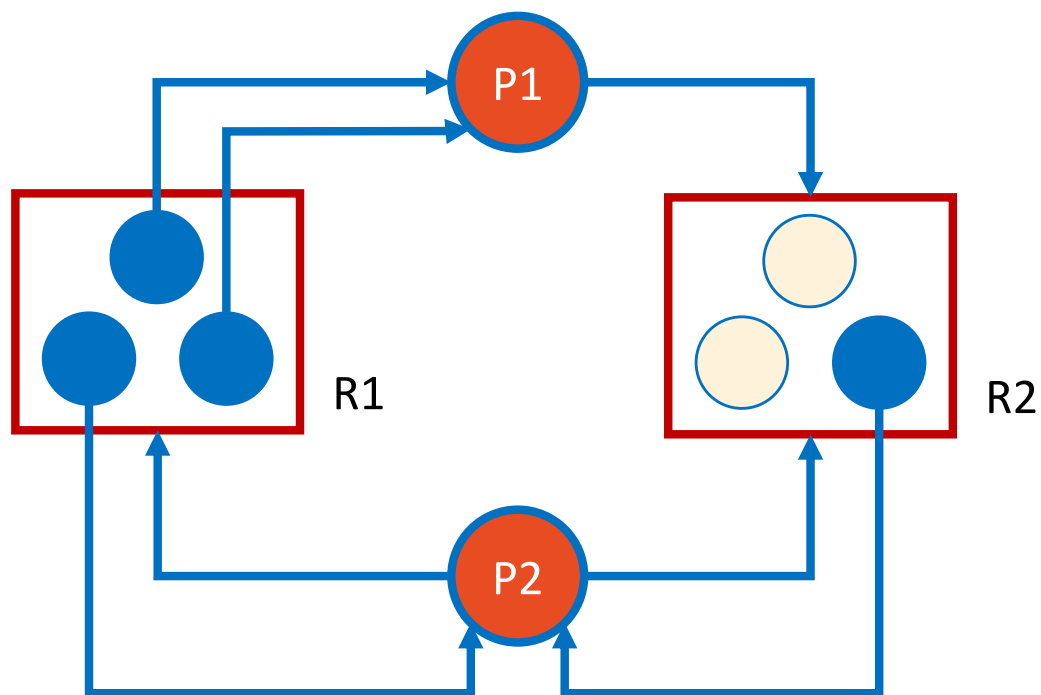
1. 从进程—资源分配图中找到一个既不阻塞又非独立的进程, 消去所有与该进程相连的有向边, 相当于该进程能够执行完成释放资源, 回收资源使之成为孤立结点. 然后将所回收的资源分配给其它进程,
2. 再从进程—资源分配图中找到下一个既不阻塞又非独立的进程, 消去所有与该进程相连的有向边, 使之成为孤立结点.....

3. 不断重复该过程, 直到图中没有既不阻塞又非独立的进程为止. 如果图中还有进程, 则系统是死锁状态. 如果图中没有进程, 则系统是可约的.

系统为死锁状态的充分条件是：当且仅当该状态的进程-资源分配图是不可完全简化的。该充分条件称为死锁定理



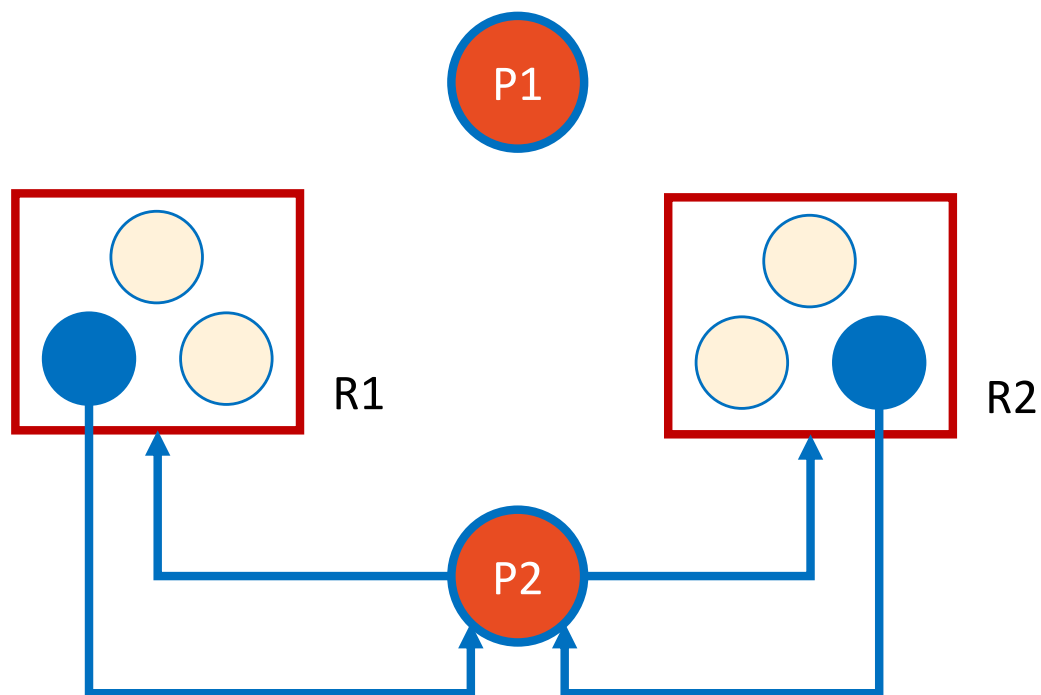
3.6.4: 简化检测方法



例1：可简化的进程资源分配图



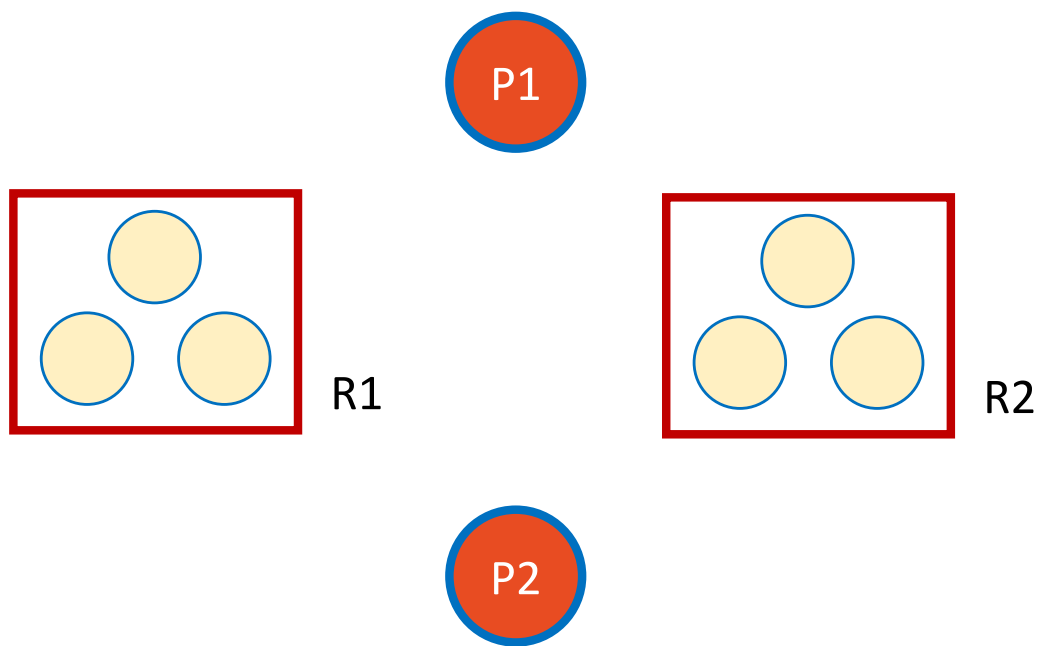
3.6.4: 简化检测方法



例1：可简化的进程资源分配图



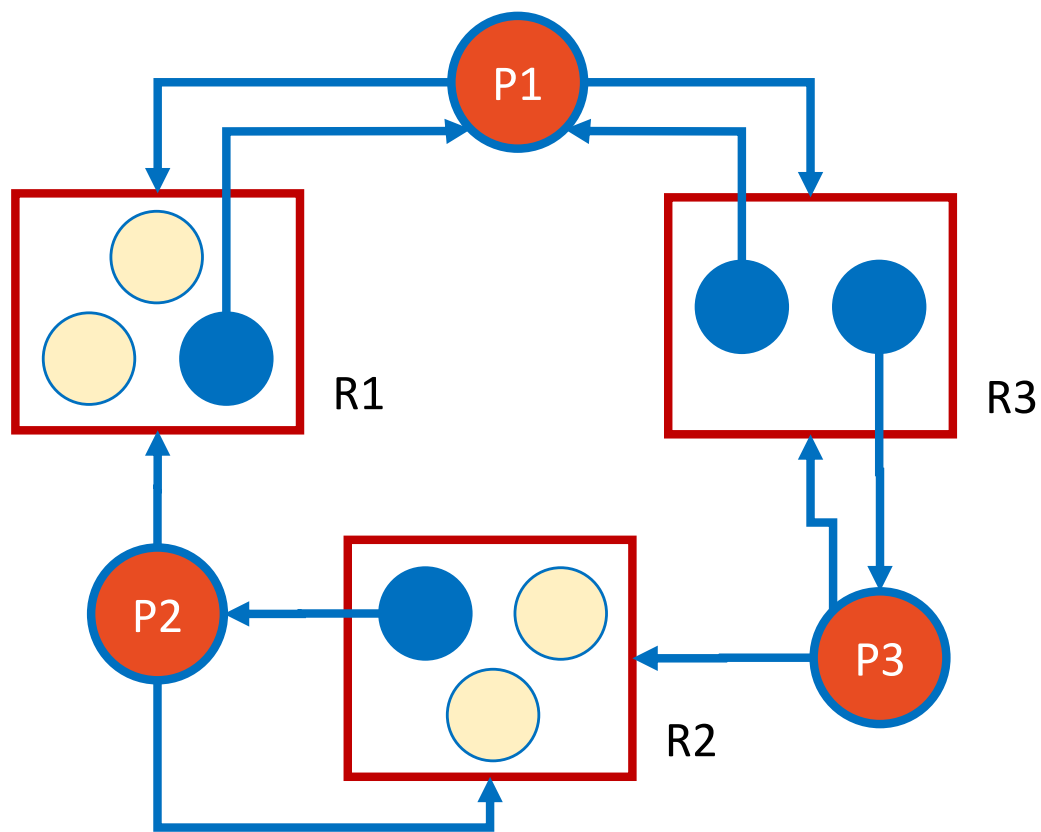
3. 6. 4: 简化检测方法



例1：可简化的进程资源分配图



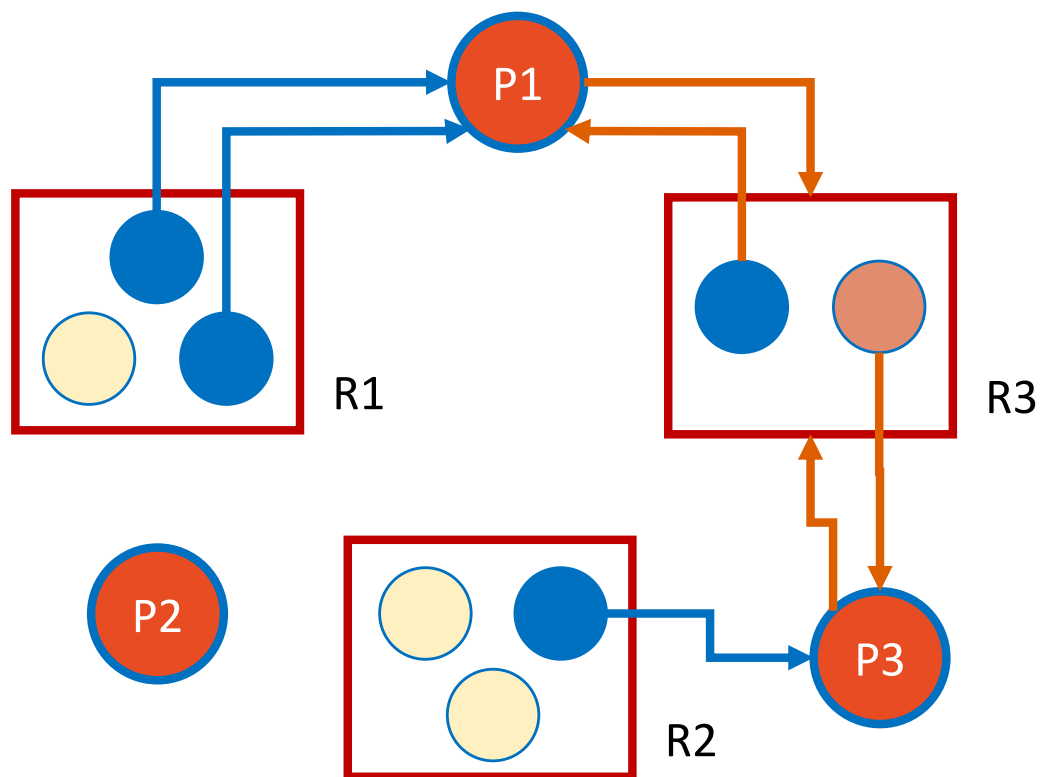
3. 6. 4: 简化检测方法



例2：不可简化的进程资源分配图



3.6.4: 简化检测方法



例2：不可简化的进程资源分配图



3.6.4: 死锁的解除

- 结束所有进程，并重启操作系统
- 撤销所有陷于死锁的进程
- 逐个撤销陷于死锁的进程，回收资源并重新分派
 - 先撤销CPU消耗时间最少的
 - 先撤销输出产生最少的
 - 先撤销预计剩余时间最长的
 - 先撤销资源分配最少的
 - 先撤销优先级最低的
- 剥夺陷于死锁进程的资源，直至死锁解除
- 进程回退至检查点（需要建立定时检查点、回退重启机制）



河海大学

计算机与信息学院

作业

- 应用题
- 2、3、6、8、20、28、42