第六章 死锁

1. **死锁 (deadlock)**
   - 一组进程由于系统资源竞争被永远阻塞，两个或多个进程对资源的访问冲突
   - 无通用的解决方案
   - 涉及到两个或者更多的进程对资源的访问冲突

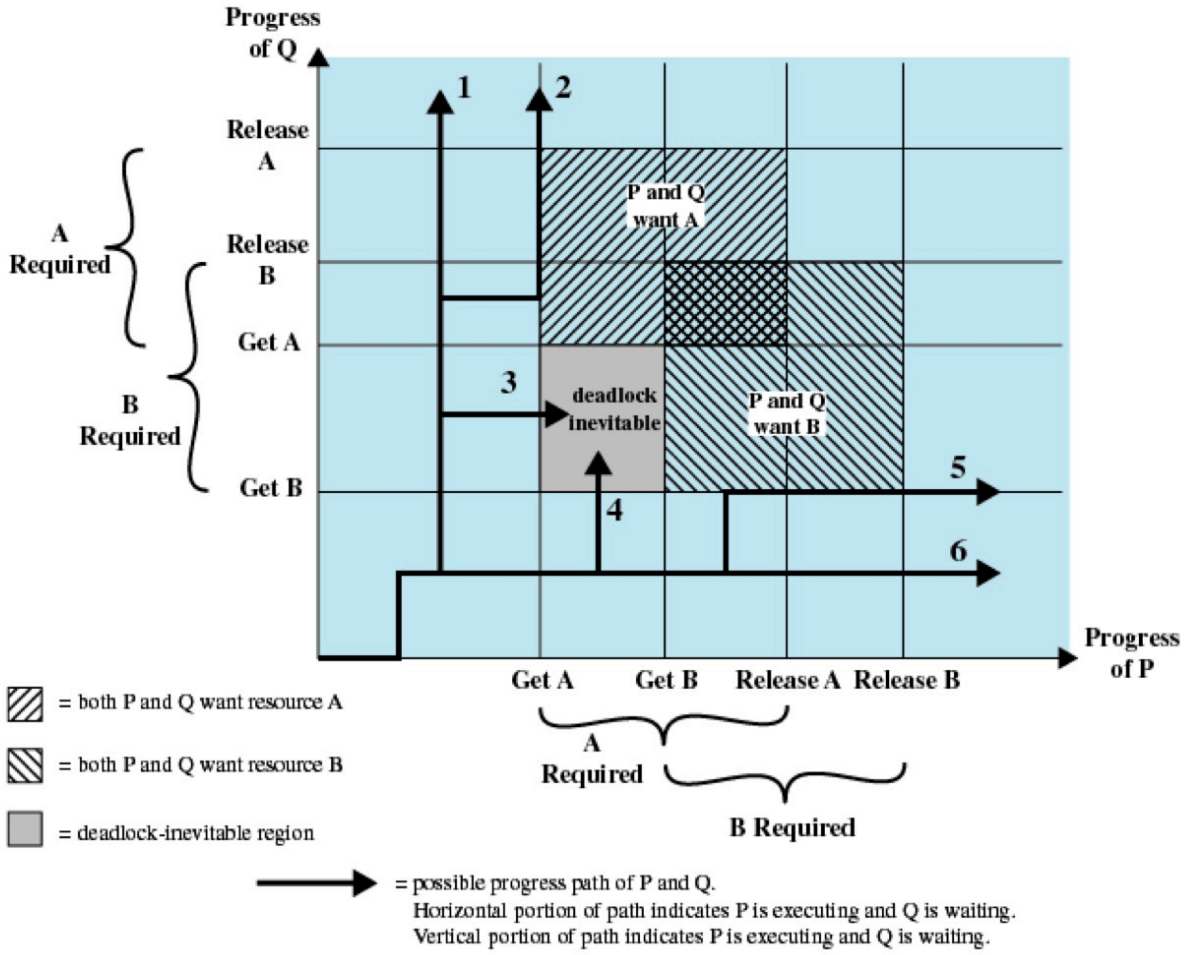


Figure 6.2 Example of Deadlock

2. 资源分配图

> 描述系统资源和进程状态的有向图

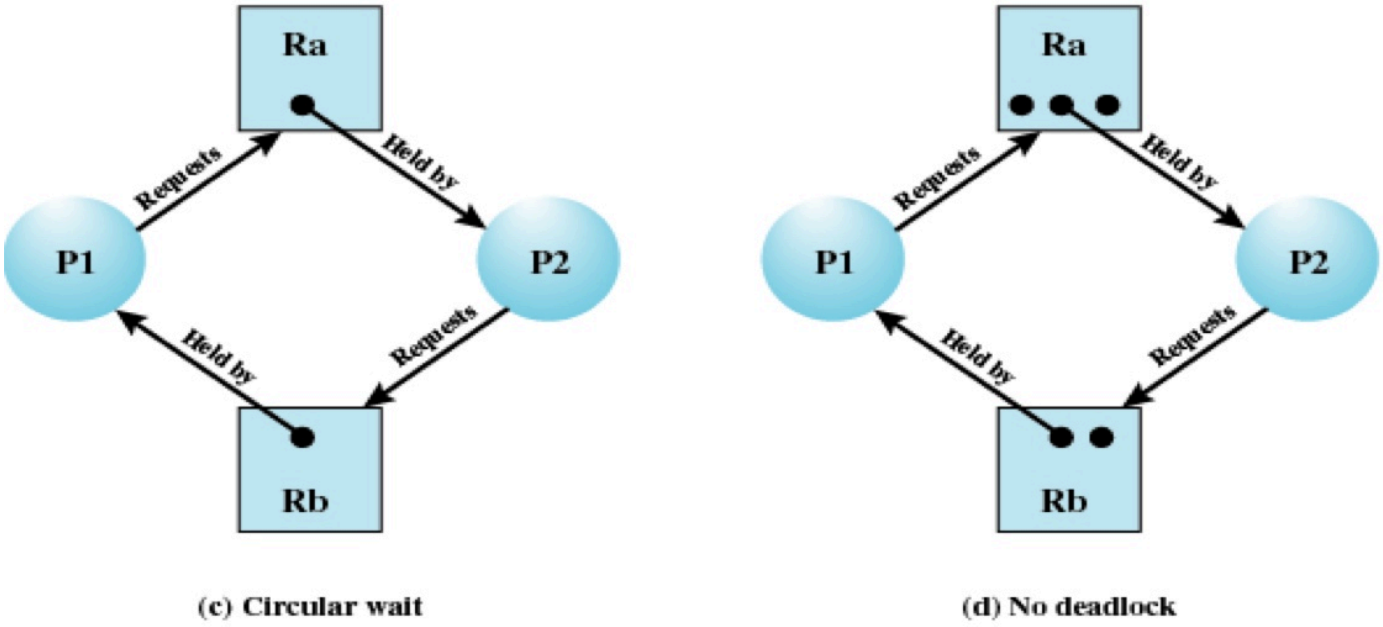


Figure 6.5 Examples of Resource Allocation Graphs

3. **死锁产生条件** 重点 背诵

   - **互斥条件**：所分配的资源排他使用
   - **不剥夺条件**：进程所获得的资源在使用完毕前无法被其他进程强行夺走
   - **请求保持条件**：在等待分配新资源的同时占有已经分配的资源
   - **循环等待条件**（*充分条件*，前三条为必要条件）：存在进程的循环等待链

4. 处理死锁的方法

   - **预防**

     > 采用某种策略消除死锁的产生条件

     - 不同策略
       - 破坏 3 个必要条件之一 —— 间接方法
       - 破坏循环等待条件 —— 直接方法
     - 互斥
       - Must be supported by the operating system
     - 一次申请所有资源
       - Require a process request all of its required resources at one time
     - 不允许进程一直持有资源

        - Process must release resource and request again
        - Operating system may preempt a process to require it releases its resources
    - 打破环路
        - Define a linear ordering of resource types

- **避免**

  > 基于资源分配的当前状态做出动态选择

    - 允许必要条件存在，但是通过选择确保永远不会达到死锁点
        - 是否允许当前的资源分配请求是动态决定的， 如果允许就有可能导致死锁
        - 需要知道未来的进程资源请求情况
    - 如果一个进程的请求会导致死锁，则不启动该进程
    - 如果一个进程增加资源的请求会导致死锁，则不允许次分配

- **检测**

  > 发现死锁的存在，并从死锁中恢复出来

    - 鸵鸟策略
    - 并发系统中的死锁现象并不是每时每刻都发生， 目前大多数操作系统在死锁问题上采用的方法是 鸵鸟算法，鸵鸟算法就是像鸵鸟一样对死锁视而 不见的算法。从数学的角度，不管花费多大代价也要防止死锁；从工程的角度，需要考虑其他因素一起进行决策，如死锁的频率，死锁的后果和危害，如果平均 1 年发生一次死锁，不管花费多 大代价也要防止死锁的决策就是错误的。如果平均 5 分种发生一次死锁，不管花费多大代价也要 防止死锁的决策就是正确的。

5. **考点***银行家算法，书P178*，资源分配算法，要求会判定安全状态

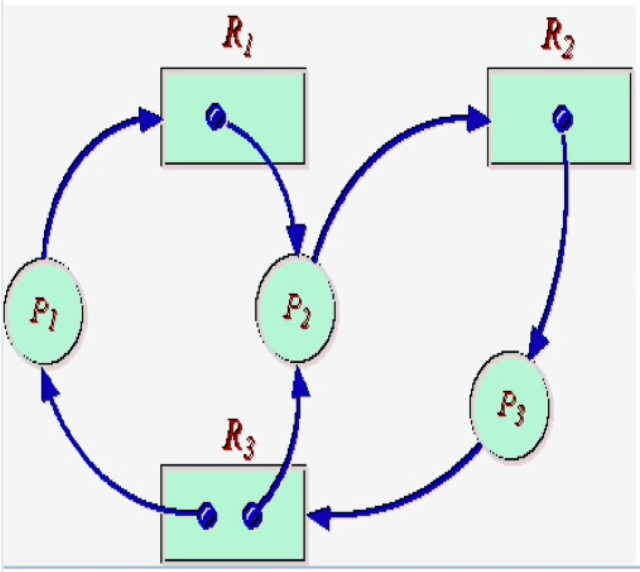
    - 事先声明每个进程请求的最大资源
    - 进程必须是无关的，执行顺序没有同步要求
    - 分配的资源数目必须是固定的
    - 在占有资源时，进程不能退出

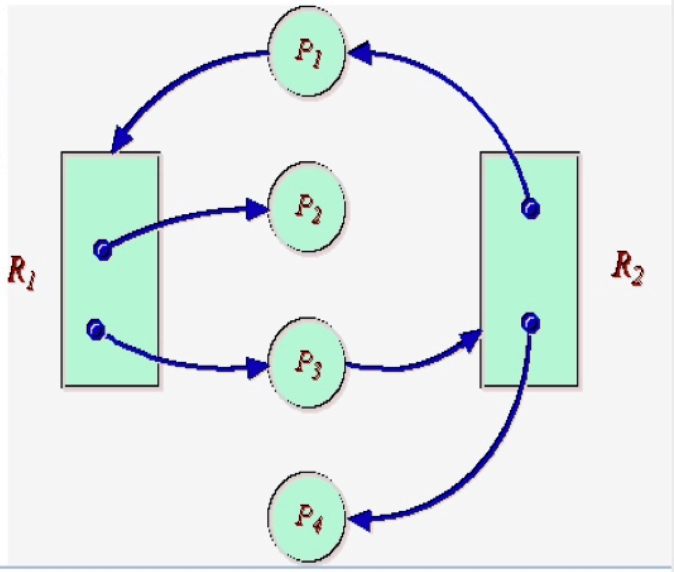
6. 检测死锁

    - 利用资源分配图

检测死锁

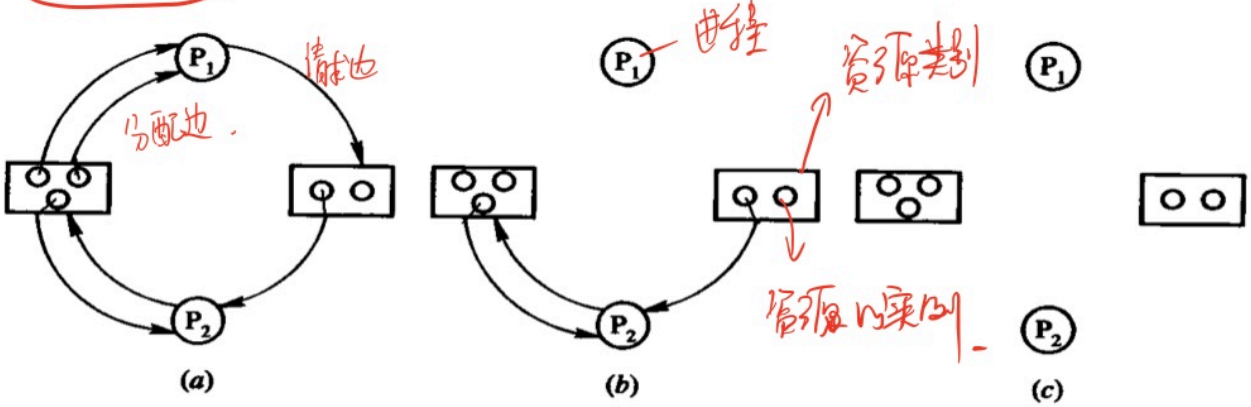
- 利用资源分配图检测
- 如果资源分配图中没有环路，则系统没有死锁；如果图中存在环路则系统中可能存在死锁。





死锁定理

- 死锁定理：S为死锁状态的充分必要条件是当且仅当S状态的资源分配图是不可化简的。
- 资源分配图的化简原则：通过对进程请求的资源进行分析，若有可用资源时，可使其请求边改为分配边。当某一进程全部是分配边没有请求边时，我们认为该进程在一有限时间内完成并将释放所有资源，因而将所有指向该进程的分配边“抹去”，成为孤立结点。这种将一个资源分配图上各个结点逐个化为孤立结点的过程称之为简化过程。



7．解除死锁

> 方法如下：

- ①系统重新启动。
- ②撤消进程。最简单的撤消方法是将所有 死锁的进程都撤销。稍微温和一些的方法 是按照某种顺序逐个的撤消进程，直至有 足够的资源可用，把死锁状态消除为止。 在出现死锁时，可采用各种策略来撤消进 程，以撤消进程所花代价最小来解除死锁 的方法是解除死锁的常用方法。

- ③剥夺资源。从其他进程中剥夺资源，以 满足死锁进程的需要，使之逐个脱离死锁 状态。 处理死锁的综合措施

> 较理想的处理死锁综合措施如下：

- ①内部资源：系统本身使用的资源。如 I/O 通道、进程 控制块，设备控制块，系统保留区等。对内部资源通过 破坏循环等待条件，即对此类资源使用有序资源分配法 预防死锁。

- ②内存资源：可以按帧或段分配给进程的存储空间。对 内存实行可剥夺式方法预防死锁是最适合的策略。当一 个进程被剥夺后，它仅仅被换到外存，释放空间以解决 死锁。

- ③进程资源：用于进程的可分配设备，如打印机、文件 等。对这类资源，死锁避免策略常常是很有效的，这是 因为进程可以事先声明他们将需要的这类资源。也可以 采用有序资源分配法预防策略。

- ④交换空间：进程交换所使用的外存交换区。通过要求 一次性分配所有请求的资源来预防死锁。也可以采用死 锁避免措施。

8. 资源分配

- 银行家算法
- 系统当前状态：当前给进程分配的资源的情况
- 安全状态是至少有一个进程执行序列不会 导致死锁

- 不安全状态

> 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

Determination of a Safe State P2 Runs to Completion

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

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) P2 runs to completion

Determination of a Safe State P1 Runs to Completion

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

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

(c) P1 runs to completion

Determination of a Safe State P3 Runs to Completion

	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**

(d) P3 runs to completion

> 不安全状态

Determination of an

Unsafe State

	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	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

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

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

(a) Initial state

Determination of an Unsafe State

	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	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
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**

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

Deadlock Avoidance Logic

```
struct state
{
        int resource[m];
        int available[m];
        int claim[n][m];
        int alloc[n][m];
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
     < error >;                                   /* total request > claim*/
else if (request [*] > available [*])
     < suspend process >;
else                                                /* simulate alloc */
{
     < define newstate by:
     alloc [i,*] = alloc [i,*] + request [*];
     available [*] = available [*] - request [*] >;
}
if (safe (newstate))
     < carry out allocation >;
else
{
     < restore original state >;
     < suspend process >;
}
```

(b) resource alloc algorithm

Deadlock Avoidance Logic

```
boolean safe (state S)
{
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible)
    {
        <find a process Pk in rest such that
            claim [k,*] - alloc [k,*] <= currentavail;>
        if (found)                              /* simulate execution of Pk */
        {
            currentavail = currentavail + alloc [k,*];
            rest = rest - {Pk};
        }
        else
            possible = false;
    }
    return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)

实例

银行家算法之例

- 如表3-4所示T0时刻的资源分配表，假定系统中有五个进程{P0，P1，P2，P3，P4}和三种类型的资源{A，B，C}，每一种资源的数量分别为10、5、7。

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

如下表所示，对T0时刻进行安全性检查，可以找到一个安全序列{P1，P3，P4，P2，P0}，系统是安全的。

进程 \ 资源情况	Work	Need	Allocation	Work+Allocation	Finish
	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
$P_$	7 4 3	4 3 1	0 0 2	7 4 5	

P_4	7 4 5	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

- ⑴**P1**发出请求**Request(1，0，2)**，执行银行家算法。
- 进行安全性检查，通过第一步和第二步检查，并找到一个安全序列**{P1，P3，P4，P2，P0}**，系统是安全的，可以分配**P1**的请求。

进程 \ 资源情况	Work A B C	Need A B C	Allocation A B C	Work+Allocation A B C	Finish
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

- ⑵**P4**发出请求**Request(3，3，0)**，执行银行家算法。
- **Available=(2，3，0)**，不能通过第二步检查（**Request[i]≤Available**），所以**P4**等待。
- ⑶**P0**请求资源，**Request（0，2，0）**，执行银行家算法。
- 进行安全性检查，通过第一步和第二步检查，如表**3-7**所示，**Available{2，1，0}**已不能满足任何进程需要，所以系统进入不安全状态，**P0**的请求不能分配。

进程 \ 资源情况	Allocation A B C	Need A B C	Available A B C

P_0	0 3 0	7 2 3	2 1 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	

银行家算法的问题