
河南大学
操作系统
计算机学院

第 3 章

处理机调度与死锁

1 3.5　死锁概述

2 3.6　预防死锁

3 3.7　避免死锁

4 3.8　死锁的检测和解除

5 哲学家进餐问题的改进解法

6 本章作业

1 3.5　死锁概述

2 3.6　预防死锁

3 3.7　避免死锁

4 3.8　死锁的检测和解除

5 哲学家进餐问题的改进解法

6 本章作业

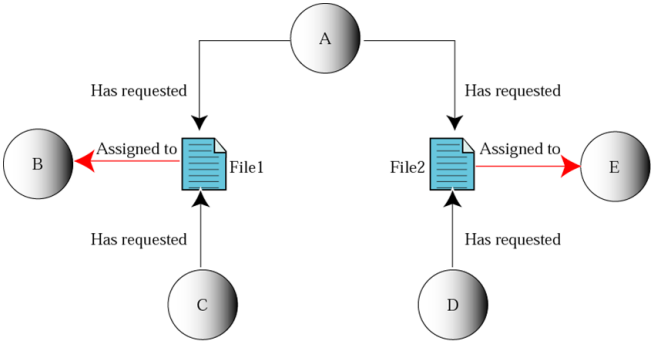
死锁概述

- 死锁: Deadlock

各并发进程彼此互相等待对方所拥有的资源，且这些并发进程在得到对方的资源之前不会释放自己所拥有的资源。从而造成大家都想得到资源而又都得不到资源，各并发进程不能继续向前推进的状态。

死锁与饥饿

- 饥饿（Starvation）指一个进程一直得不到资源。
- 死锁和饥饿都是由于进程竞争资源而引起的。饥饿一般不占有资源，死锁进程一定占有资源。



资源的类型

1 可重用资源和消耗性资源

- 可重用资源（永久性资源）
  可被多个进程多次使用，如所有硬件。
  - 只能分配给一个进程使用，不允许多个进程共享。
  - 进程在对可重用资源的使用时，须按照请求资源、使用资源、释放资源这样的顺序。
  - 系统中每一类可重用资源中的单元数目是相对固定的，进程在运行期间，既不能创建，也不能删除。

资源的类型

- 消耗性资源（临时性资源）

  是在进程运行期间，由进程动态的创建和消耗的。

  - 消耗性资源在进程运行期间是可以不断变化的，有时可能为 0；
  - 进程在运行过程中，可以不断地创造可消耗性资源的单元，将它们放入该资源类的缓冲区中，以增加该资源类的单元数目。
  - 进程在运行过程中，可以请求若干个可消耗性资源单元，用于进程自己消耗，不再将它们返回给该资源类中。
  - 可消耗资源通常是由生产者进程创建，由消费者进程消耗。最典型的可消耗资源是用于进程间通信的消息。

资源的类型

2 可抢占资源和不可抢占资源

- 可抢占资源
  - 可抢占资源指某进程在获得这类资源后，该资源可以再被其他进程或系统抢占。对于这类资源是不会引起死锁的。
  - CPU 和内存均属于可抢占性资源。
- 不可抢占资源
  - 一旦系统把某资源分配给该进程后，就不能将它强行收回，只能在进程用完后自行释放。
  - 打印机等属于不可抢占性资源。

计算机系统中的死锁

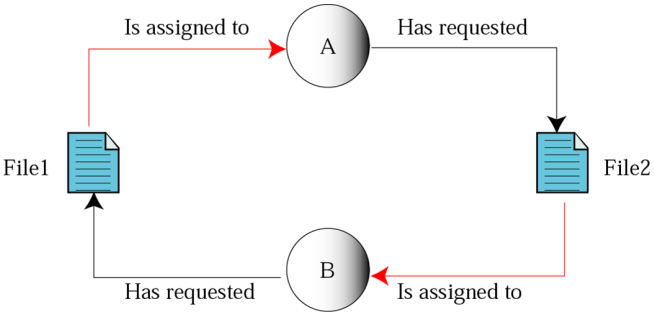
死锁是多个进程在运行过程中因争夺资源而造成的一种僵局（Deadly-Embrace)），若无外力作用，这些进程都将无法向前推进。

死锁产生的原因

1. 竞争不可抢占资源引起死锁
2. 竞争可消耗资源引起死锁
3. 进程间推进顺序不当引起死锁

计算机系统中的死锁

1 竞争不可抢占资源引起死锁



计算机系统中的死锁

2 竞争可消耗资源引起死锁（以进程间通信为例）

```
顺序1：
P1：…
    send（p2，m1）；
    receive（p3, m3）；
    …
P2：
    …
    send（p3，m2）；
    receive（p1, m1）；
    …
P3：
    …
    send（p1，m3）；
    receive（p2, m2）；
    …
```

计算机系统中的死锁

2 竞争可消耗资源引起死锁（以进程间通信为例）

顺序1：

```
P1 : ...
    send ( p2 , m1 ) ;
    receive ( p3, m3 ) ;
    ...
P2 :
    ...
    send ( p3 , m2 ) ;
    receive ( p1, m1 ) ;
    ...
P3 :
    ...
    send ( p1 , m3 ) ;
    receive ( p2, m2 ) ;
    ...
```

顺序2：

```
P1 : ...
    receive ( p3, m3 ) ;
    send ( p2 , m1 ) ;
    ...
P2 :
    ...
    receive ( p1, m1 ) ;
    send ( p3 , m2 ) ;
    ...
P3 :
    ...
    receive ( p2, m2 ) ;
    send ( p1 , m3 ) ;
    ...
```

计算机系统中的死锁

2 竞争可消耗资源引起死锁（以进程间通信为例）

顺序1：

```
P1 : ...
    send ( p2 , m1 ) ;
    receive ( p3, m3 ) ;
    ...
P2 :
    ...
    send ( p3 , m2 ) ;
    receive ( p1, m1 ) ;
    ...
P3 :
    ...
    send ( p1 , m3 ) ;
    receive ( p2, m2 ) ;
    ...
```

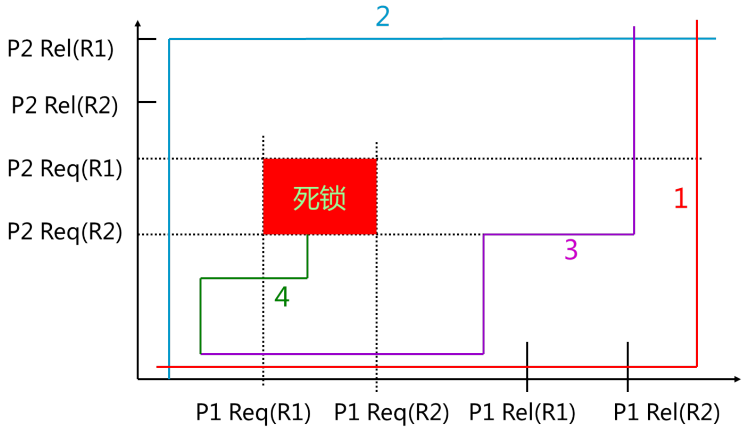
顺序2：

```
P1 : ...
    receive ( p3, m3 ) ;
    send ( p2 , m1 ) ;
    ...
P2 :
    ...
    receive ( p1, m1 ) ;
    send ( p3 , m2 ) ;
    ...
P3 :
    ...
    receive ( p2, m2 ) ;
    send ( p1 , m3 ) ;
    ...
```

死锁

计算机系统中的死锁

3 进程间推进顺序不当引起死锁



死锁的定义

- 死锁的定义：

死锁是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态，这些永远在互相等待的进程称为死锁进程。

- 关于死锁的一些结论：
  1. 参与死锁的进程数至少为两个
  2. 参与死锁的所有进程均等待资源
  3. 参与死锁的进程至少有两个已经占有资源
  4. 死锁进程是系统中当前进程集合的一个子集
- 死锁会浪费大量系统资源，甚至导致系统崩溃。

产生死锁的必要条件

产生死锁的四个必要条件

1. 互斥条件（mutual exclusion）
2. 请求和保持条件（hold-while-applying）
3. 不可剥夺条件（non preemption）
4. 环路等待条件（circular wait）

产生死锁的四个必要条件

1. 互斥条件：进程对所分配到的资源进行排它性使用。
2. 请求和保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放。
3. 不可剥夺条件：进程已获得的资源，在未使用完之前不能被抢占，只能在进程使用完时由自己释放。
4. 环路等待条件：指在发生死锁时，必然存在一个进程 -资源的循环链，即进程集合 P0，P1，P2，...，Pn 中的 P0，正在等待一个 P1 占用的资源，P1 正在等待 P2 占用的资源，.....，Pn 正在等待已被 P0 占用的资源。

处理死锁的方法

1. 鸵鸟方法：对死锁视而不见，即忽略死锁。
2. 预防死锁：通过设置某些限制条件，去破坏产生死锁的四个必要条件中的一个或几个条件，来防止死锁的发生。
3. 避免死锁：在资源的动态分配过程中，用某种方法去防止系统进入不安全状态，从而避免死锁的发生。
4. 检测死锁：允许系统在运行过程中发生死锁，但可设置检测机构及时检测死锁的发生，并采取适当措施加以清除。
5. 解除死锁：当检测出死锁后，便采取适当措施将进程从死锁状态中解脱出来。

1 3.5　死锁概述

2 3.6　预防死锁

3 3.7　避免死锁

4 3.8　死锁的检测和解除

5 哲学家进餐问题的改进解法

6 本章作业

预防死锁

破坏死锁的四个必要条件中的一个或几个。

预防死锁

1. 破坏互斥条件

   即允许多个进程同时访问资源。但由于资源本身固有特性的限制，此方法不可行。

2. 破坏请求和保持条件
3. 破坏不可剥夺条件
4. 破坏环路等待条件

破坏请求和保持条件

1 第一种协议

- 全分配，全释放: 采用预先静态分配方法，即要求进程在运行之前一次性申请它所需要的全部资源，在它的资源未满足前，不把它投入运行。
- 若系统有足够的资源，便可把进程需要的所有资源分配给它，在整个运行期间便不会再提出资源要求，从而摒弃了请求条件。在该进程的等待期间，它并未占有任何资源，因而也摒弃了保持条件，这样可以保证系统不会发生死锁。
- 简单安全，但资源浪费严重，可能有饥饿现象，同时必须预知进程所需要的全部资源。

破坏请求和保持条件

2 第二种协议

- 允许一个进程只获得运行初期所需的资源后，便开始运行。
- 摒弃保持条件：进程运行过程中必须释放已分配给自己的且已经用完的全部资源，然后才能再请求新的所需资源。
- 第二种协议能使进程更快地完成任务，提高设备利用率，还可减少进程饥饿的几率。

破坏不可剥夺条件

1 实施方案 1

- 在允许进程动态申请资源前提下，规定一个进程在申请新的资源不能立即得到满足而变为等待状态之前，必须释放已占有的全部资源，若需要再重新申请。

2 实施方案 2

- 进程申请的资源被其他进程占用时，可以通过操作系统抢占这一资源（适用于状态易于保存和恢复的资源）。

- 破坏不可剥夺条件的方法实现较复杂，需付出很大的代价。会增加系统开销，降低系统吞吐量，且进程前段工作可能失效。

破坏环路等待条件

- 采用有序资源分配方法，即将系统中所有资源都按类型赋予一个编号，要求每个进程均严格按照资源序号递增的次序来请求资源, 从而保证任何时刻的资源分配图不出现环路。
- 例如：系统把所有资源按类型进行排队。如输入机 =1，打印机 =2，磁带机 =3，磁盘机 =4。
- 如果一个进程已经分配了序号为 i 资源，它接下来请求的资源只能是那些排在 i 之后的资源。
- 当 i<j 时，进程 A 获得 Ri，可以请求 Rj，而进程 B 获得 Rj 再请求 Ri不可能发生，因为资源 Ri 排在 Rj 前面。

破坏环路等待条件

- 在采用这种策略时，总有一个进程占据了较高序号的资源，此后它继续申请的资源必然是空闲的，因而进程可以一直向前推进。
- 有序资源分配方法的缺点：新增资源不便（原序号已排定），用户不能自由申请资源，加重了进程负担。使用资源顺序与申请顺序不同可能造成资源浪费。

1 3.5　死锁概述

2 3.6　预防死锁

3 3.7　避免死锁

4 3.8　死锁的检测和解除

5 哲学家进餐问题的改进解法

6 本章作业

避免死锁

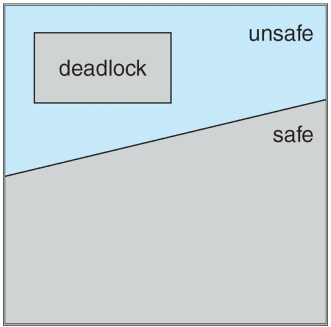
- 不需要事先采取限制措施破坏产生死锁的必要条件（死锁预防）；在资源的动态分配过程中，采用某种策略防止系统进入不安全状态，从而避免发生死锁。
- 在系统运行过程中，对进程发出的每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源，若分配后系统可能发生死锁，则不予分配，否则予以分配。

系统的安全状态

- 安全状态指在某一时刻，系统能按某种进程顺序 (p1,p2,...，pn) 来为每个进程 Pi 分配其资源，直到满足每个进程对资源的最大需求，使每个进程都可顺利地完成，则称此时的系统状态为安全状态，称序列 (p1,p2,...，pn) 为安全序列。若某一时刻系统中不存在这样一个安全序列，则称此时的系统状态为不安全状态。
- 在死锁避免的方法中，允许进程动态申请资源，系统在进行资源分配之前，先计算资源分配的安全性，若此次分配不会导致系统进入不安全状态，便将资源分配给进程，否则进程等待。

死锁状态空间

- 如果一个系统处于安全状态，就不会死锁。
- 如果一个系统处于不安全状态，就有可能死锁。
- 避免死锁的实质：确保系统不进入不安全状态。



安全状态实例

例：假定系统中有三个进程 P1、P2 和 P3，共有 12 台打印机，三个进程对打印机的需求和占有情况如下表所示：

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

- T0 时刻，存在一个安全序列（P2，P1，P3），所以系统是安全的。

安全序列可能不唯一

由安全状态向不安全状态的转换

上例中，若 P3 再申请一台，则进入不安全状态。

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

- 在 P3 请求资源时，尽管系统中尚有可用的打印机，但却不能分配给它，必须让 P3 一直等待到 P1 和 P2 完成，释放出资源后再将足够的资源分配给 P3。

银行家算法（Banker's Algorithm）

- 银行家算法是最具代表性的避免死锁算法（Dijkstra 于 1965 年提出）
- 银行家算法的实质：设法保证系统动态分配资源后不进入不安全状态，以避免可能产生的死锁。
- 检查是否有足够的剩余资金满足一个距最大请求最近的客户。如果有，这笔贷款被认为是能够收回的。继续检查下一个客户，如果所有投资最终都被收回，那么该状态是安全的，最初的请求可以批准。
- 银行家算法执行的前提条件：要求进程必须预先提出自己的最大资源请求数量，这一数量不能超过系统资源的总量，系统资源的总量是一定的。

银行家算法中的数据结构

假定系统中有 n 个进程（P1，P2，...，Pn），m 类资源（R1，R2，...，Rm），银行家算法中使用的数据结构如下：

- 可利用资源向量：Available[j]=k，表示 Rj类资源有 k 个可用
- 最大需求矩阵：Max[i,j]=k，进程 Pi最大请求 k 个 Rj类资源
- 分配矩阵：Allocation[i,j]=k，进程 Pi已经分配到 k 个 Rj类资源
- 需求矩阵：Need[i,j]=k，进程 Pi还需要 k 个 Rj 类资源

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

银行家算法描述（资源分配算法）

假定进程 Pi 请求分配 Rj 类资源 k 个，设 Request[i,j]=k。当进程 Pi 发出资源请求后，系统按如下步骤进行检查：

1. 如果 Request[i,j] ≤Need[i,j] ，转 (2)；否则出错，因为进程申请资源量超过它声明的最大量。
2. 如果 Request[i,j] ≤Available[j] ，转 (3)；否则表示资源不够，需等待。
3. 系统试分配资源给进程 Pi，并作如下修改：
   Available[j]= Available[j] - Request[i,j]
   Allocation[i,j]= Allocation[i,j] + Request[i,j]
   Need[i,j]= Need[i,j] - Request[i,j]
4. 系统执行安全性算法。若安全，则正式进行分配，否则恢复原状态让进程 Pi 等待。

银行家算法描述（安全性算法）

为了进行安全性检查，需要定义如下数据结构：

1. 工作变量 work[m]：记录可用资源。开始时，Work= Available。
2. finish[n]：记录系统是否有足够的资源分配给进程，使之运行完成。开始时，finish[i]=false；当有足够资源分配给进程 Pi 时，令 finish[i]=true。

银行家算法描述（安全性算法）

1. Work := Available; Finish [i] = false
2. 寻找满足如下条件的进程 Pi
   ① Finish [i] = false
   ② Need[i,j]≤ Work[j] ，如果找到，转 (3)，否则转 (4)
3. 当进程 Pi 获得资源后，可顺利执行完，并释放分配给它的资源，故执行：
   Work[j] := Work[j] + Allocation[i,j] ；Finish[i] := true ，转 (2)
4. 若所有进程的 Finish [i] = true ，则表示系统处于安全状态，否则处于不安全状态。

银行家算法例题

例：假定系统中有 5 个进程 P0 到 P4，3 类资源及数量分别为 A(10 个），B（5 个），C（7 个），T0 时刻的资源分配情况如下：

	Max	Allocation	Need	Available
	A B C	A B C	A B C	A B C
P0	7 5 3	0 1 0	7 4 3	3 3 2
P1	3 2 2	2 0 0	1 2 2	
P2	9 0 2	3 0 2	6 0 0	
P3	2 2 2	2 1 1	0 1 1	
P4	4 3 3	0 0 2	4 3 1	

（1）T0 时刻的安全性

Available
A B C
3 3 2

	Work A B C	Need A B C	Allocation A B C	Work+ Allocation A B C	Finish
P0		7 4 3	0 1 0		false
P1		1 2 2	2 0 0		false
P2		6 0 0	3 0 2		false
P3		0 1 1	2 1 1		false
P4		4 3 1	0 0 2		false

（1）T0 时刻的安全性

Available
A B C
3 3 2

	Work A B C	Need A B C	Allocation A B C	Work+ Allocation A B C	Finish
P0		7 4 3	0 1 0		false
P1	3 3 2	1 2 2	2 0 0	5 3 2	true
P2		6 0 0	3 0 2		false
P3		0 1 1	2 1 1		false
P4		4 3 1	0 0 2		false

（1）T0 时刻的安全性

Available
A B C
3 3 2

	Work A B C	Need A B C	Allocation A B C	Work+ Allocation A B C	Finish
P0		7 4 3	0 1 0		false
P1	3 3 2	1 2 2	2 0 0	5 3 2	true
P2		6 0 0	3 0 2		false
P3	5 3 2	0 1 1	2 1 1	7 4 3	true
P4		4 3 1	0 0 2		false

（1）T0 时刻的安全性

Available
A B C
3 3 2

	Work A B C	Need A B C	Allocation A B C	Work+ Allocation A B C	Finish
P0		7 4 3	0 1 0		false
P1	3 3 2	1 2 2	2 0 0	5 3 2	true
P2		6 0 0	3 0 2		false
P3	5 3 2	0 1 1	2 1 1	7 4 3	true
P4	7 4 3	4 3 1	0 0 2	7 4 5	true

（1）T0 时刻的安全性

Available
A B C
3 3 2

	Work A B C	Need A B C	Allocation A B C	Work+ Allocation A B C	Finish
P0		7 4 3	0 1 0		false
P1	3 3 2	1 2 2	2 0 0	5 3 2	true
P2	7 4 5	6 0 0	3 0 2	10 4 7	true
P3	5 3 2	0 1 1	2 1 1	7 4 3	true
P4	7 4 3	4 3 1	0 0 2	7 4 5	true

（1）T0 时刻的安全性

Available
A B C
3 3 2

	Work A B C	Need A B C	Allocation A B C	Work+ Allocation A B C	Finish
P0	10 4 7	7 4 3	0 1 0	10 5 7	true
P1	3 3 2	1 2 2	2 0 0	5 3 2	true
P2	7 4 5	6 0 0	3 0 2	10 4 7	true
P3	5 3 2	0 1 1	2 1 1	7 4 3	true
P4	7 4 3	4 3 1	0 0 2	7 4 5	true

(2) P1 请求资源 Request1(1,0,2)

P1 发出请求向量 Request1(1,0,2)

已知 Need1 (1,2,2)，Available (3,3,2)

系统按银行家算法进行检查：

1. Request1 (1,0,2) $\leq$ Need1 (1,2,2)
2. Request1 (1,0,2) $\leq$ Available (3,3,2)
3. 系统试为 P1 分配资源，并修改相应的向量 Available、Need、Allocation

(2) P1 请求资源 Request1(1,0,2)

修改相应的向量 Available、Need、Allocation

	Max	Allocation	Need	Available
	A B C	A B C	A B C	A B C
P0	7 5 3	0 1 0	7 4 3	3 3 2
P1	3 2 2	2 0 0	1 2 2	
P2	9 0 2	3 0 2	6 0 0	
P3	2 2 2	2 1 1	0 1 1	
P4	4 3 3	0 0 2	4 3 1	

(2) P1 请求资源 Request1(1,0,2)

修改相应的向量 Available、Need、Allocation

	Max A B C	Allocation A B C	Need A B C	Available A B C
P0	7 5 3	0 1 0	7 4 3	3 3 2 **(2 3 0)**
P1	3 2 2	2 0 0 **(3 0 2)**	1 2 2 **(0 2 0)**	
P2	9 0 2	3 0 2	6 0 0	
P3	2 2 2	2 1 1	0 1 1	
P4	4 3 3	0 0 2	4 3 1	

(2) P1 请求资源 Request1(1,0,2)

初始：Work = Available = [2 3 0]

	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		0 1 1	2 1 1		false
P_4		4 3 1	0 0 2		false
P_2		6 0 0	3 0 2		false
P_0		7 4 3	0 1 0		false

(2) P1 请求资源 Request1(1,0,2)

初始：Work = Available = [2 3 0]

	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		4 3 1	0 0 2		false
P_2		6 0 0	3 0 2		false
P_0		7 4 3	0 1 0		false

(2) P1 请求资源 Request1(1,0,2)

初始：Work = Available = [2 3 0]

	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_2		6 0 0	3 0 2		false
P_0		7 4 3	0 1 0		false

(2) P1 请求资源 Request1(1,0,2)

初始：Work = Available = [2 3 0]

	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_2	7 4 5	6 0 0	3 0 2	10 4 7	true
P_0		7 4 3	0 1 0		false

(2) P1 请求资源 Request1(1,0,2)

初始：Work = Available = [2 3 0]

	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_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

由安全性检查分析得知：此时刻存在着一个安全序列 P1,P3,P4,P2,P0，故系统是安全的，可以立即将 P1 所申请的资源分配给它。

（3）P4 请求资源 Request4(3,3,0)

P4 发出请求向量 Request4(3,3,0)，系统按银行家算法进行检查：

1. Request4(3,3,0) $\leq$ Need4 (4,3,1)
2. Request4 (3,3,0) $>$ Available (2,3,0)

资源不够，让 P4 等待。

(4)P0 请求资源 Request0(0,2,0)

P0 发出请求向量 Request0(0,2,0)，系统按银行家算法进行检查：

1. Request0(0,2,0) $\leq$ Need0 (7,4,3)
2. Request0 (0,2,0) $\leq$ Available (2,3,0)

系统试为 P0 分配资源，并修改相应的向量

Available、Need、Allocation

(4)P0 请求资源 Request0(0,2,0)

	Max	Allocation	Need	Available
	A B C	A B C	A B C	A B C
P0	7 5 3	0 1 0	7 4 3	3 3 2
				(2 3 0)
P1	3 2 2	2 0 0	1 2 2	
		(3 0 2)	(0 2 0)	
P2	9 0 2	3 0 2	6 0 0	
P3	2 2 2	2 1 1	0 1 1	
P4	4 3 3	0 0 2	4 3 1	

(4)P0 请求资源 Request0(0,2,0)

	Max A B C	Allocation A B C	Need A B C	Available A B C
P0	7 5 3	0 1 0	7 4 3	3 3 2
				(2 3 0)
		[0 3 0]	[7 2 3]	[2 1 0]
P1	3 2 2	2 0 0	1 2 2	
		(3 0 2)	(0 2 0)	
P2	9 0 2	3 0 2	6 0 0	
P3	2 2 2	2 1 1	0 1 1	
P4	4 3 3	0 0 2	4 3 1	

Avaliable(2,1,0) 已不能满足任何进程需要，故系统进入不安全状态，此时系统不分配资源。

银行家算法练习题

例：T0 时刻的资源分配情况如下：

	最大资源需求			已分配资源数量		
	A	B	C	A	B	C
P1	5	5	9	2	1	2
P2	5	3	6	4	0	2
P3	4	0	11	4	0	5
P4	4	0	5	2	0	4
P5	4	2	4	3	1	4

剩余资源向量为：Available =（2, 3, 3）

1. T0 时刻是否为安全状态？若是，请给出安全序列。
2. 在 T0 时刻若进程 P2 请求资源（0，3，4），是否能实施资源分配？

银行家算法练习题　剩余向量 = （2, 3, 3）

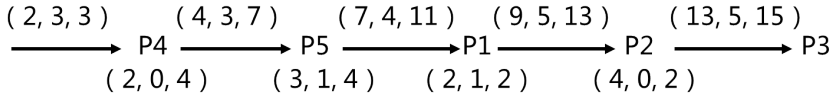
	最大资源需求Max			已分配资源数量			尚需资源Need		
	A	B	C	A	B	C	A	B	C
P1	5	5	9	2	1	2	3	4	7
P2	5	3	6	4	0	2	1	3	4
P3	4	0	11	4	0	5	0	0	6
P4	4	0	5	2	0	4	2	0	1
P5	4	2	4	3	1	4	1	1	0

1. T0 时刻是否为安全状态？若是，请给出安全序列。

银行家算法练习题　剩余向量 = （2, 3, 3）

	最大资源需求Max			已分配资源数量			尚需资源Need		
	A	B	C	A	B	C	A	B	C
P1	5	5	9	2	1	2	3	4	7
P2	5	3	6	4	0	2	1	3	4
P3	4	0	11	4	0	5	0	0	6
P4	4	0	5	2	0	4	2	0	1
P5	4	0	2	3	0	1	1	1	0

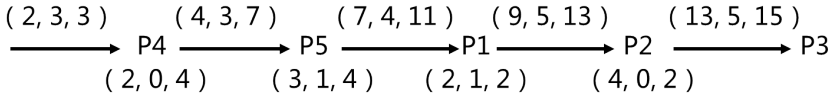
1 T0 时刻是否为安全状态？若是，请给出安全序列。



银行家算法练习题　剩余向量 = （2, 3, 3）

	最大资源需求Max			已分配资源数量			尚需资源Need		
	A	B	C	A	B	C	A	B	C
P1	5	5	9	2	1	2	3	4	7
P2	5	3	6	4	0	2	1	3	4
P3	4	0	11	4	0	5	0	0	6
P4	4	0	5	2	0	4	2	0	1
P5	4	0	2	3	0	1	1	1	0

1 T0 时刻是否为安全状态？若是，请给出安全序列。



T0 时刻为安全状态，安全序列：p4, p5, p1, p2, p3

银行家算法练习题　剩余向量 = （2, 3, 3）

	最大资源需求Max			已分配资源数量			尚需资源Need		
	A	B	C	A	B	C	A	B	C
P1	5	5	9	2	1	2	3	4	7
P2	5	3	6	4	0	2	1	3	4
P3	4	0	11	4	0	5	0	0	6
P4	4	0	5	2	0	4	2	0	1
P5	4	0	2	4	1	4	1	1	0

2 在 T0 时刻若进程 P2 请求资源（0，3，4），是否能实施资源分配？

银行家算法练习题　剩余向量 =（2, 3, 3）

	最大资源需求Max			已分配资源数量			尚需资源Need		
	A	B	C	A	B	C	A	B	C
P1	5	5	9	2	1	2	3	4	7
P2	5	3	6	4	0	2	1	3	4
P3	4	0	11	4	0	5	0	0	6
P4	4	0	5	2	0	4	2	0	1
P5	4	0	2	3	1	4	1	1	0

2 在 T0 时刻若进程 P2 请求资源（0，3，4），是否能实施资源分配？

因为 Request2（0, 3, 4）> Available （2, 3, 3）

所以不能满足进程 P2 的请求。

银行家算法练习题

	最大资源需求Max			已分配资源数量			尚需资源Need		
	A	B	C	A	B	C	A	B	C
P1	5	5	9	2	1	2	3	4	7
P2	5	3	6	4	0	2	1	3	4
P3	4	0	11	4	0	5	0	0	6
P4	4	0	5	2	0	4	2	0	1
P5	4	0	4	3	1	4	1	1	0

3 若进程 P4 请求资源（2，0，0），是否可以资源分配？

银行家算法练习题

	最大资源需求Max			已分配资源数量			尚需资源Need		
	A	B	C	A	B	C	A	B	C
P1	5	5	9	2	1	2	3	4	7
P2	5	3	6	4	0	2	1	3	4
P3	4	0	11	4	0	5	0	0	6
P4	4	0	5	2	0	4	2	0	1
P5	4	0	2	3	1	4	1	1	0

3 若进程 P4 请求资源（2，0，0），是否可以资源分配？

Request4（2, 0, 0）≤ Need4 (2, 0, 1)

且 Request4（2, 0, 0）< Available（2, 3, 3）

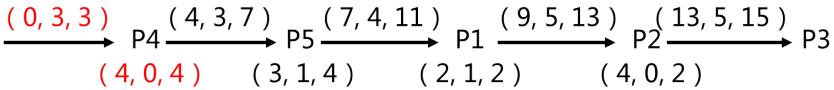
试分配后，则 Need4 =(0, 0, 1), Allocation4 =(4, 0, 4)

新的剩余向量 Available =（0, 3, 3）

银行家算法练习题

	最大资源需求 Max			已分配资源数量			尚需资源Need		
	A	B	C	A	B	C	A	B	C
P1	5	5	9	2	1	2	3	4	7
P2	5	3	6	4	0	2	1	3	4
P3	4	0	11	4	0	5	0	0	6
P4	4	0	5	4	0	4	0	0	1
P5	4	2	4	3	1	4	1	1	0

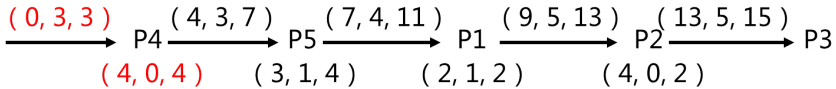
新的剩余向量 Available = （0, 3, 3）



银行家算法练习题

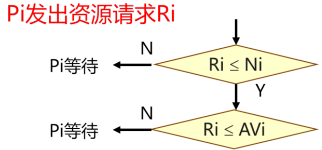
	最大资源需求 Max			已分配资源数量			尚需资源Need		
	A	B	C	A	B	C	A	B	C
P1	5	5	9	2	1	2	3	4	7
P2	5	3	6	4	0	2	1	3	4
P3	4	0	11	4	0	5	0	0	6
P4	4	0	5	4	0	4	0	0	1
P5	4	2	4	3	1	4	1	1	0

新的剩余向量 Available = （0, 3, 3）



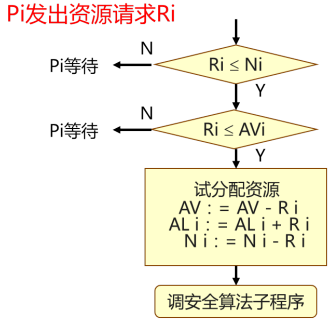
系统安全，安全序列：p4, p5, p1, p2, p3 ，可以分配

银行家算法流程图



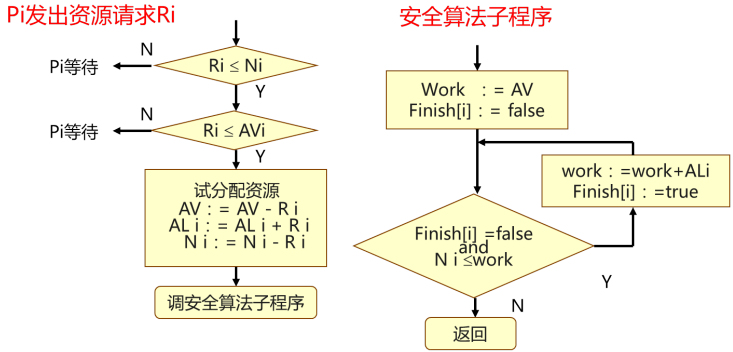
Work　　工作向量，表示当前系统可
　　提供给进程的各类资源数。
Finish [i]　True / False
AV= Available，AL= Allocation，N=Need

银行家算法流程图



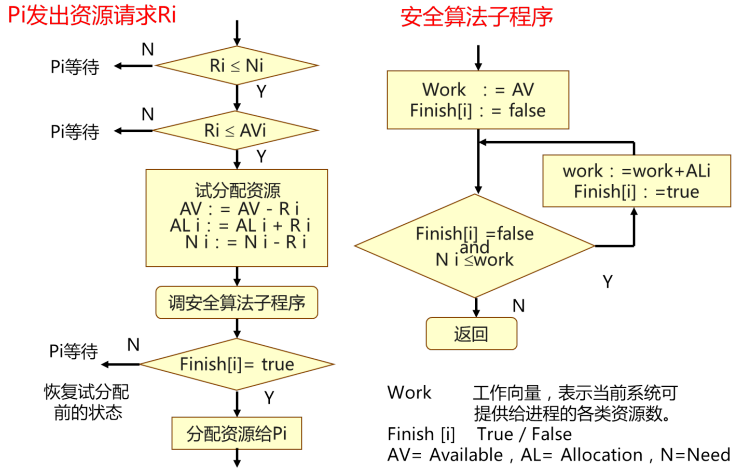
Work　　工作向量，表示当前系统可
　　提供给进程的各类资源数。
Finish [i]　True / False
AV= Available，AL= Allocation，N=Need

银行家算法流程图



Work　　工作向量，表示当前系统可提供给进程的各类资源数。
Finish [i]　True / False
AV= Available，AL= Allocation，N=Need

银行家算法流程图



Work　　工作向量，表示当前系统可提供给进程的各类资源数。

Finish [i]　True / False

AV= Available，AL= Allocation，N=Need

死锁避免小结

- 优点
  - 比死锁预防限制少。
  - 无死锁检测方法中的资源剥夺，进程重启。
- 缺点
  - 必须事先声明每个进程的最大资源请求。
  - 考虑的进程必须是无关的，也就是说，它们执行的顺序没有任何同步要求的限制。
  - 进程的数量保持固定不变，且分配的资源数目必须是固定的。

1 3.5　死锁概述

2 3.6　预防死锁

3 3.7　避免死锁

4 3.8　死锁的检测和解除

5 哲学家进餐问题的改进解法

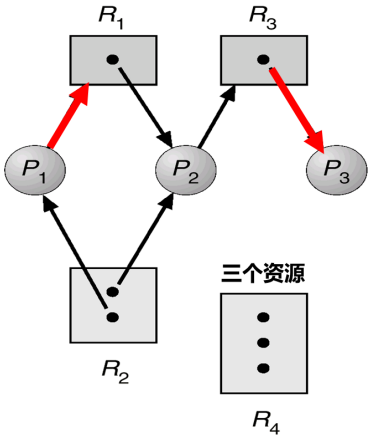
6 本章作业

死锁的检测和解除

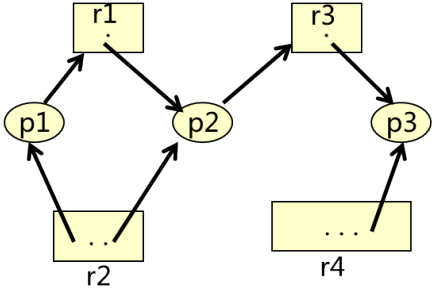
- 如果在一个系统中，既未采用死锁预防方法，也未采用死锁避免方法，而是直接为进程分配资源，则系统中便有可能发生死锁。
- 一旦死锁发生，系统应能将其找到并加以消除，为此需提供死锁检测和解除死锁的手段。
- 检测死锁的基本思想：在操作系统中保存资源的请求和分配信息，利用某种算法对这些信息加以检查，以判断是否存在死锁。

资源分配图

- 资源分配图又称进程 -资源图，它是描述进程和资源间的申请和分配关系的一种有向图。
- 圆圈表示进程节点 P，方框表示资源节点 R，方框中的小黑点数表示资源数。
- 请求边：Pi →Rj
  分配边：Pi ←Rj

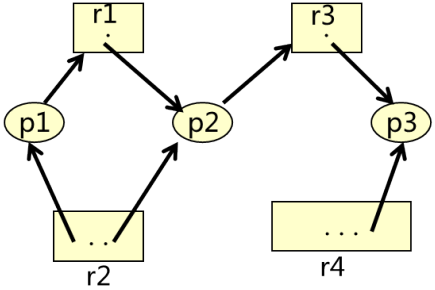


资源分配图



是否死锁？执行顺序？

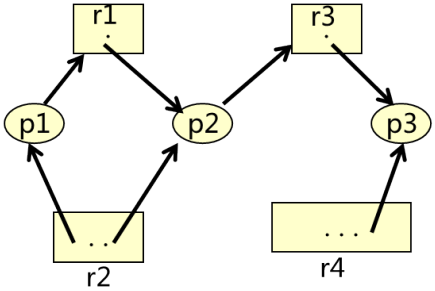
资源分配图



是否死锁？执行顺序？

无环路，无死锁

资源分配图

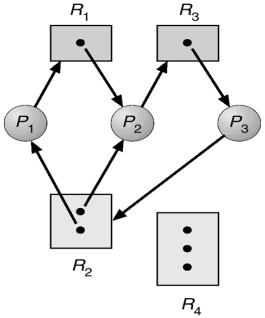


是否死锁？执行顺序？

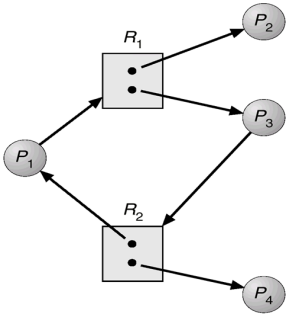
无环路，无死锁

执行顺序 P3,P2,P1

资源分配图

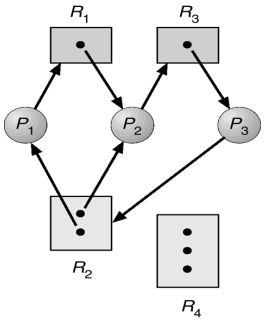


有环路有死锁

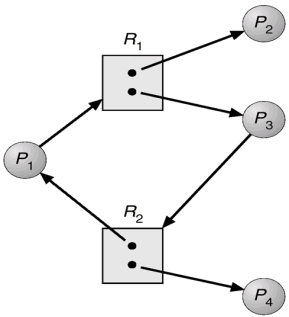


有环路无死锁

资源分配图



有环路有死锁



有环路无死锁

重要结论：如果资源分配图中不存在环路，则系统中不存在死锁；反之，如果资源分配图中存在环路，则系统中可能存在死锁，也可能不存在死锁。

死锁定理

资源分配图的化简方法

1. 寻找一个既不阻塞又不孤立的进程结点 Pi，若无则算法结束；
2. 去除 Pi 的所有分配边和请求边，使 Pi 成为一个孤立结点；
3. 转步骤（1）。

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；反之，称该图是不可完全简化的。

- 孤立结点：没有请求边和分配边与之相连。
- 阻塞结点：有请求边但资源无法满足其要求。

死锁定理

资源分配图的化简方法

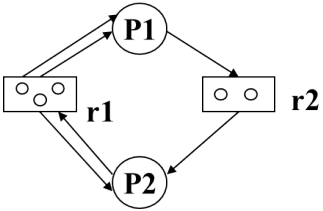
1. 寻找一个既不阻塞又不孤立的进程结点 Pi，若无则算法结束；
2. 去除 Pi 的所有分配边和请求边，使 Pi 成为一个孤立结点；
3. 转步骤（1）。

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；反之，称该图是不可完全简化的。

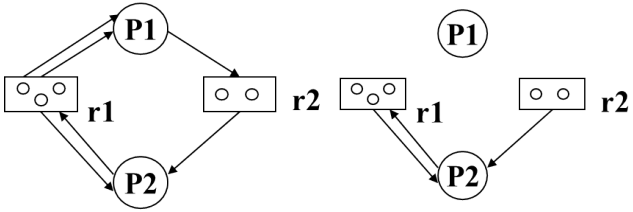
- 孤立结点：没有请求边和分配边与之相连。
- 阻塞结点：有请求边但资源无法满足其要求。

死锁定理:
出现死锁状态的充分条件是资源分配图不可完全简化。

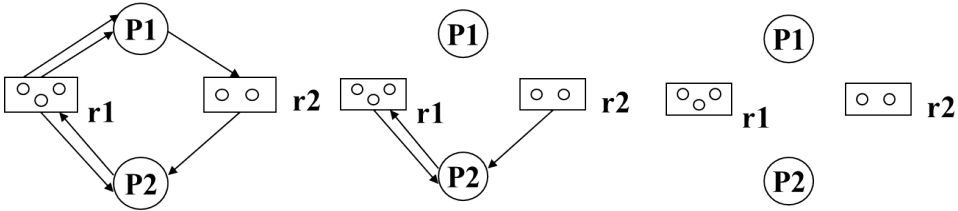
死锁定理



死锁定理

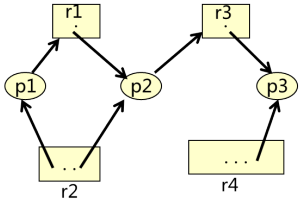


死锁定理

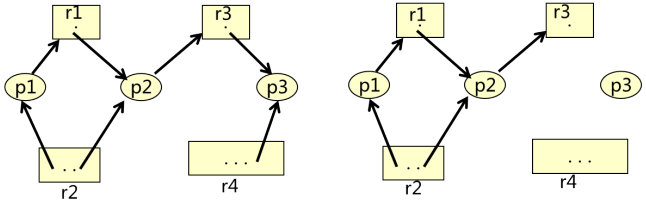


资源分配图可完全简化

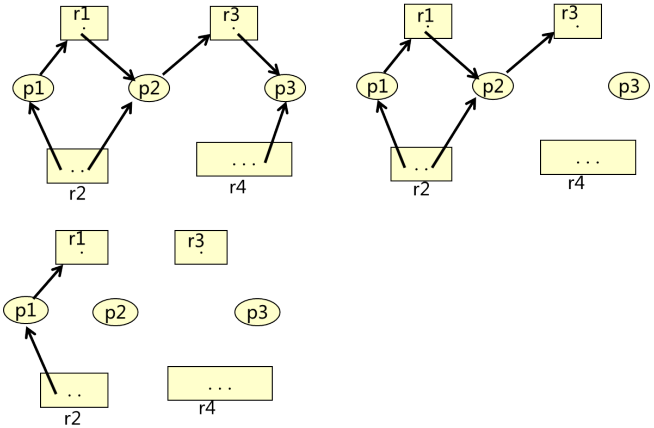
死锁定理



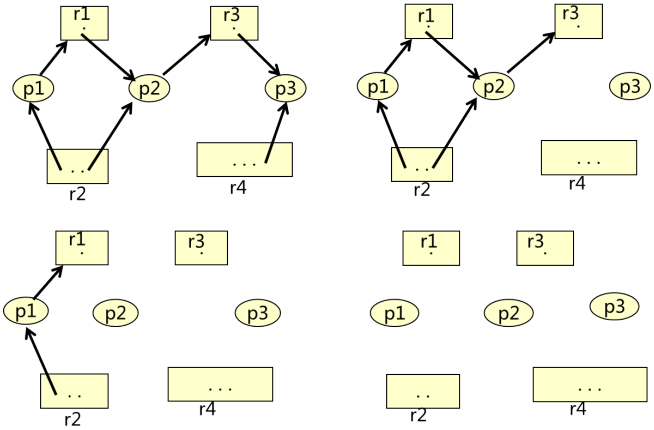
死锁定理



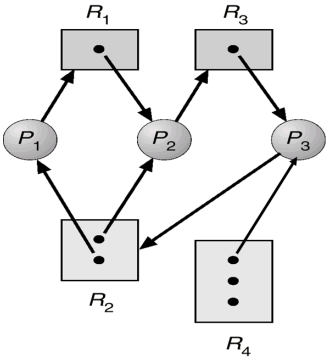
死锁定理



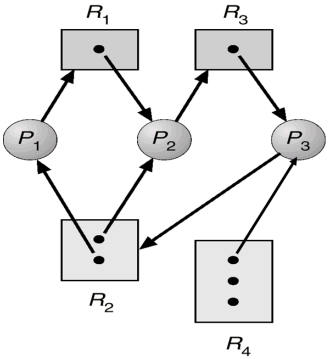
死锁定理


r1
r3
p1
p2
p3
r2
r4
r1
r3
p1
p2
p3
r2
r4
r1
r3
p1
p2
p3
r2
r4
r1
r3
p1
p2
p3
r2
r4

死锁定理



死锁定理



该图是不可完全简化的。三个进程结点均为阻塞结点。

资源分配图的化简结果与化简顺序无关，最终结果相同。

死锁的检测

- 死锁检测算法的思想
  - 死锁检测算法的思想是基于资源分配图化简和死锁定理来检测死锁。
- 死锁检测的原因
  - 系统没有采取任何预先限制死锁的措施。资源分配时不检查系统是否会进入不安全状态, 被请求的资源都被授予给进程。需要周期性检测是否出现死锁。
- 检测时机
  - 在每个资源请求时都进行（相当于死锁避免）。
  - 定时检测。
  - 系统资源利用率下降时检测死锁。

死锁的检测

死锁检测中的数据结构类似于银行家算法的数据结构：

Request请求矩阵

	R1	R2	...	R_m
P1	0	1	....	
P2	1	0		
.....	...			
.....	...			
Pn				

Allocation资源分配矩阵

	R1	R2	...	R_m
P1	2	0	...	
P2	1	2		
.....	...			
.....	...			
Pn				

L进程向量

P1	
P2	
.....	...
.....	...
Pn	

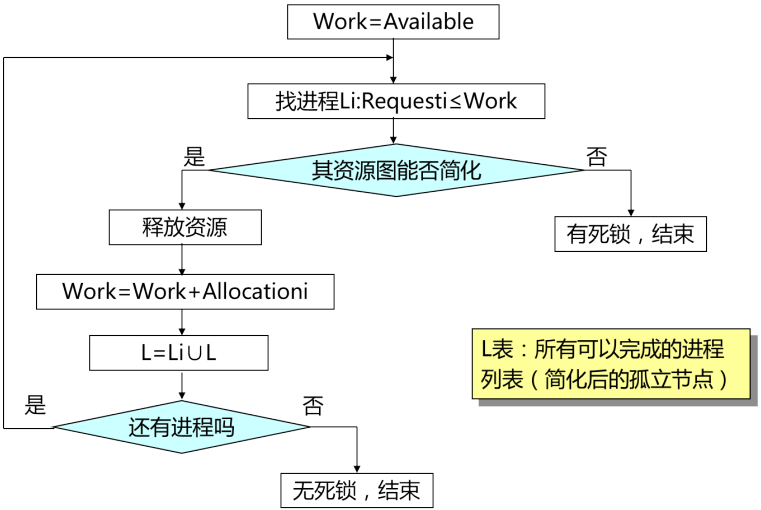
Available可用资源向量

R1	R2	...	Rm
0	1	...	

Work工作向量

R1	R2	...	Rm
0	1	...	

死锁检测流程



L表：所有可以完成的进程列表（简化后的孤立节点）

检测算法练习题

五个进程P_0到P_4，　三个资源类型：A（7个），B（2个），C（6个）

时刻T_0的状态

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

<P_0, P_2, P_3, P_1, P_4> 会导致对所有i，　*Finish*[*i*] = true.

检测算法练习题

假设P_2 又请求了一个 C资源.

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

系统状态?

没有剩余的C类资源。P_0可以归还所有的资源，但是资源不够完成其他任何进程的请求。

死锁存在，包括进程P_1, P_2, P_3, P_4

死锁的解除

- 一旦检测出系统中出现了死锁，就应将陷入死锁的进程从死锁状态中解脱出来，常用的解除死锁方法有两种:
- 资源剥夺法: 当发现死锁后，从其它进程剥夺足够数量的资源给死锁进程，以解除死锁状态。
- 撤消进程法: 采用强制手段从系统中撤消一个/一部分死锁进程，并剥夺这些进程的资源供其它死锁进程使用。
  - 撤消全部死锁进程。
  - 按照某种顺序逐个地撤消进程，直至有足够的资源可用，使死锁状态消除为止。

死锁的解除

- 基于最小代价原则一次只终止一个进程直到取消死锁循环为止。
- 撤消进程选择原则
  - 已消耗 CPU 时间最少
  - 到目前为止产生的输出量最少
  - 预计剩余的时间最长
  - 目前为止分配的资源总量最少
  - 优先级最低

1 3.5　死锁概述

2 3.6　预防死锁

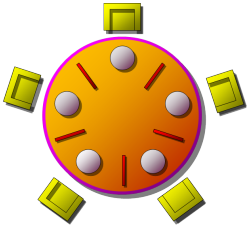
3 3.7　避免死锁

4 3.8　死锁的检测和解除

5 哲学家进餐问题的改进解法

6 本章作业

“哲学家进餐”问题



- 有五个哲学家，他们的生活方式是交替地进行思考和进餐。他们共用一张圆桌，分别坐在五张椅子上。
- 在圆桌上有五个碗和五支筷子，平时一个哲学家进行思考，饥饿时便试图取用其左、右最靠近他的筷子，只有在他拿到两支筷子时才能进餐。进餐完毕，放下筷子又继续思考。

哲学家进餐问题可看作是并发进程并发执行时处理共享资源的一个有代表性的问题。

“哲学家进餐”问题

```
Var chopstick: array[0，…，4] of semaphore=1；
/*5支筷子分别设置为初始值为1的互斥信号量*/
```

第i个哲学家的活动

```
repeat
    wait(chopstick[i])；
    wait(chopstick[(i+1) mod 5])；
        eat；
    signal(chopstick[i])；
    signal(chopstick[(i+1) mod 5])；
        think；
until false；
```

“哲学家进餐”问题

- 此算法可以保证不会有相邻的两位哲学家同时进餐。
- 若五位哲学家同时饥饿而各自拿起了左边的筷子，这使五个信号量 chopstick 均为 0，当他们试图去拿起右边的筷子时，都将因无筷子而无限期地等待下去，即可能会引起死锁。

哲学家进餐问题的改进解法

- 方法一：至多只允许四位哲学家同时去拿左筷子，最终能保证至少有一位哲学家能进餐，并在用完后释放两只筷子供他人使用。
- 方法二：仅当哲学家的左右手筷子都拿起时才允许进餐。
- 方法三：规定奇数号哲学家先拿左筷子再拿右筷子，而偶数号哲学家相反。

哲学家进餐问题的改进解法

- 方法一：至多只允许四位哲学家同时去拿左筷子，最终能保证至少有一位哲学家能进餐，并在用完后释放两只筷子供他人使用。
  - 设置一个初值为 4 的信号量 r，只允许 4 个哲学家同时去拿左筷子，这样就能保证至少有一个哲学家可以就餐，不会出现饿死和死锁的现象。
  - 原理：至多只允许四个哲学家同时进餐，以保证至少有一个哲学家能够进餐，最终总会释放出他所使用过的两支筷子，从而可使更多的哲学家进餐。

哲学家进餐问题的改进解法

```
semaphore chopstick[5]={1，1，1，1，1};
semaphore r=4;
void philosopher(int i)
{
   while(true)
   {
      think();
      wait(r);                                  //请求进餐
      wait(chopstick[i]);                       //请求左手边的筷子
      wait(chopstick[(i+1) mod 5]);             //请求右手边的筷子
      eat();
      signal(chopstick[(i+1 ) mod 5]);          //释放右手边的筷子
      signal(chopstick[i]);                     //释放左手边的筷子
      signal(r);                                //释放信号量r
      think();
   }
}
```

哲学家进餐问题的改进解法

- 方法二：仅当哲学家的左右手筷子都拿起时才允许进餐。
  - 解法 1：利用 AND 型信号量机制实现。
  - 原理：多个临界资源，要么全部分配，要么一个都不分配，因此不会出现死锁的情形。

哲学家进餐问题的改进解法

```
semaphore chopstick[5]={1，1，1，1，1};
void philosopher(int i)
{
   while(true)
      {
         think();
         Swait(chopstick[i]; chopstick[(i+1) mod 5]);
         eat();
         Ssignal(chopstick[(i+1) mod 5]; chopstick[i]);
         think();
      }
}
```

哲学家进餐问题的改进解法

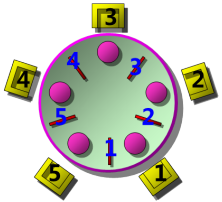
- 方法二：仅当哲学家的左右手筷子都拿起时才允许进餐。
  - 解法 2：利用信号量的保护机制实现。
  - 原理：通过互斥信号量 mutex 对 eat() 之前取左侧和右侧筷子的操作进行保护，可以防止死锁的出现。

哲学家进餐问题的改进解法

```
semaphore mutex = 1;
semaphore chopstick[5]={1 , 1 , 1 , 1 , 1};
void philosopher(int i)
{
   while(true)
      {
            think();
         wait(mutex);
         wait(chopstick[i]);
         wait(chopstick [(i+1) mod 5]);
         signal(mutex);
            eat();
         signal(chopstick [(i+1) mod 5]);
         signal(chopstick[i]);
             think();
      }
}
```

哲学家进餐问题的改进解法

- 方法三：规定奇数号哲学家先拿左筷子再拿右筷子，而偶数号哲学家相反。
- 原理：按照下图，将是 2,3 号哲学家竞争 3 号筷子，4,5 号哲学家竞争 5 号筷子。1 号哲学家不需要竞争。最后总会有一个哲学家能获得两支筷子而进餐。



哲学家进餐问题的改进解法

```
semaphore chopstick[5]={1，1，1，1，1};
void philosopher(int i)
{
    while(true)
        {
            if(i mod 2 == 0)                    //偶数哲学家，先右后左。
                {
                    wait (chopstick [(i + 1) mod 5]);
                    wait (chopstick [i]) ;
                    eat();
                    signal (chopstick [i]);
                    signal (chopstick[(i+1) mod 5]);
                }
            Else                                //奇数哲学家，先左后右。
                {
                    wait (chopstick [i]) ;
                    wait (chopstick [(i+1) mod 5]);
                    eat();
                    signal (chopstick [(i+1) mod 5]);
                    signal (chopstick [i]);
                }
        }
}
```

1 3.5　死锁概述

2 3.6　预防死锁

3 3.7　避免死锁

4 3.8　死锁的检测和解除

5 哲学家进餐问题的改进解法

6 本章作业

本章作业

谢 谢

School of Computer & Information Engineering
Henan University
Kaifeng, Henan Province
475001
China